List 5.36 shows the notification queue, i.e., the thread that processes the new incoming chatting messages concurrently. The received chatting messages are retained on the server until they are polled by relevant partners. Again, the code is the application of the pattern of DWC for notifications. Table 5.22 summarizes it. You need to make a comparison between that one and the one in Table 5.20. Remember the pattern and try to employ it when processing incoming notifications on the server side.

```
1    package com.greatfree.chat.server;
2
3    import com.greatfree.chat.PrivateChatSessions;
4    import com.greatfree.chat.message.cs.ChatNotification;
5    import com.greatfree.concurrency.NotificationQueue;
6    import com.greatfree.testing.data.ServerConfig;
7
8    /*
9     * The thread deals with chatting notification from the client. 04/23/2017, Bing Li
10    */
11
12   // Created: 04/27/2017, Bing Li
13   public class ChatThread extends NotificationQueue<ChatNotification>
14   {
15       public ChatThread(int taskSize)
16       {
17           super(taskSize);
18       }
19
20       @Override
21       public void run()
22       {
23           ChatNotification notification;
24           while (!this.isShutdown())
25           {
26               while (!this.isEmpty())
27               {
28                   try
29                   {
30                       notification = this.getNotification();
31
32                       // When new messages are available, they are retained in the server for
33                       // polling. 05/25/2017, Bing Li
34                       PrivateChatSessions.HUNGARY().addMessage(notification.getSessionKey(),
35                           notification.getSenderKey(), notification.getReceiverKey(), notification.getMessage());
36                       this.disposeMessage(notification);
37                   }
38                   catch (InterruptedException e)
39                   {
40                       e.printStackTrace();
41                   }
42               }
43               try
44               {
45                   // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
46                   this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
47               }
48               catch (InterruptedException e)
49               {
50                   e.printStackTrace();
51               }
52           }
53       }
54   }
```

List 5.36 The code of ChatThread.java

List 5.37 shows the code of ChatThreadCreator.java, which creates instances of the notification queue when no sufficient threads are available.

```
1    package com.greatfree.chat.server;
2
3    import com.greatfree.chat.message.cs.ChatNotification;
4    import com.greatfree.concurrency.NotificationThreadCreatable;
5
6    // Created: 04/27/2017, Bing Li
7    public class ChatThreadCreator implements NotificationThreadCreatable<ChatNotification, ChatThread>
8    {
9        @Override
10       public ChatThread createNotificationThreadInstance(int taskSize)
11       {
12           return new ChatThread(taskSize);
13       }
14   }
```

List 5.37 The code of ChatThreadCreator.java

Pattern: DWC (Double While Concurrency)		
Class	ChatThread	
Singleton	No	
Method	void dispose()	
	void enqueue(ChatNotification notification)	
	ChatNotification getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	ChatNotification
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 5.22 The pattern of DWC to process incoming chatting notifications

2.6.3 The ND for Shutdown Chattering Server Notification

The last case to use the ND is the one for the notification, ShutdownChatServerNotification. It first emerges in Line 63 of the code of ChatManServerDispatcher.java in List 5.10. Table 5.23 makes a conclusion for the ND that dispatches the shutdown notification.

Pattern: ND (Notification Dispatcher)		
Class	NotificationDispatcher<ShutdownChatServerNotification, ShutdownChattingServerThread, ShutdownChattingServerThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(ShutdownChatServerNotification notification)	
Enclosed Pattern	Notification	ShutdownChatServerNotification
	NQ (Notification Queue)	ShutdownChattingServerThread
	NQC (Notification Queue Creator)	ShutdownChattingServerThreadCreator
Employed API	ServerMessage	
	NotificationQueue<Notification extends ServerMessage>	
	NotificationThreadCreatable<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>>	

Table 5.23 The pattern of ND to dispatch the shutdown notifications

The notification of ShutdownChatServerNotification is shown in List 5.38.

```
1    package com.greatfree.chat.message;
2
3    import com.greatfree.message.ServerMessage;
4
5    /*
```

```
6      * The notification aims to notify the chatting server to shutdown. 01/20/2016, Bing Li
7      */
8
9      // Created: 04/17/2017, Bing Li
10     public class ShutdownChatServerNotification extends ServerMessage
11     {
12         private static final long serialVersionUID = -7450693179089567 32L;
13
14         public ShutdownChatServerNotification()
15         {
16             super(ChatMessageType.SHUTDOWN_CHAT_SERVER_NOTIFICATION);
17         }
18     }
```

List 5.38 The code of ShutdownChatServerNotification.java

The thread for the ND is called ShutdownChattingServerThread, which is shown in List 5.39. When the shutting-down notification is received, it stops the chatting server by calling the method of stop() of the chatting sever in Line 42 of the code, ChatServer, in List 5.2. This is also an application of the pattern of DWC, as summarized in Table 5.24.

Pattern: DWC (Double While Concurrency)		
Class	ShutdownChattingServerThread	
Singleton	No	
Method	void dispose()	
	void enqueue(ShutdownChatServerNotification notification)	
	ShutdownChatServerNotification getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	ShutdownChatServerNotification
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 5.24 The pattern of DWC to process the shutdown notifications

```
1      package com.greatfree.chat.server;
2
3      import java.io.IOException;
4
5      import com.greatfree.chat.message.ShutdownChatServerNotification;
6      import com.greatfree.concurrency.NotificationQueue;
7      import com.greatfree.exceptions.RemoteReadException;
8      import com.greatfree.testing.data.ServerConfig;
9      import com.greatfree.util.ServerStatus;
10
11     /*
12      * The thread deals with shutting down notification from the administrator. 04/23/2017, Bing Li
13      */
14
15     // Created: 04/18/2017, Bing Li
16     public class ShutdownChattingServerThread extends NotificationQueue<ShutdownChatServerNotification>
17     {
18         public ShutdownChattingServerThread(int taskSize)
19         {
20             super(taskSize);
21         }
22
23         @Override
24         public void run()
25         {
26             ShutdownChatServerNotification notification;
27             while (!this.isShutdown())
28             {
29                 while (!this.isEmpty())
30                 {
31                     try
```

```
32                  {
33                      notification = this.getNotification();
34                      ServerStatus.FREE().setShutdown();
35                      ChatServer.CS().stop();
36                      this.disposeMessage(notification);
37                  }
38                  catch (InterruptedException | IOException | ClassNotFoundException |
39                      RemoteReadException e)
40                  {
41                      e.printStackTrace();
42                  }
43              }
44              try
45              {
46                  // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
47                  this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
48              }
49              catch (InterruptedException e)
50              {
51                  e.printStackTrace();
52              }
53          }
54      }
55  }
```

List 5.39 The code of ShutdownChattingServerThread.java

The last component for the ND is the thread creator, ShutdownChattingServerThreadCreator, as shown in List 5.40. The same as others, the creator generates instances of the thread, ShutdownChattingServerThread, when no enough threads are available.

```
1   package com.greatfree.chat.server;
2
3   import com.greatfree.chat.message.ShutdownChatServerNotification;
4   import com.greatfree.concurrency.NotificationThreadCreatable;
5
6   // Created: 04/18/2017, Bing Li
7   public class ShutdownChattingServerThreadCreator implements NotificationThreadCreatable
8       <ShutdownChatServerNotification, ShutdownChattingServerThread>
9   {
10      @Override
11      public ShutdownChattingServerThread createNotificationThreadInstance(int taskSize)
12      {
13          return new ShutdownChattingServerThread(taskSize);
14      }
15  }
```

List 5.40 The code of ShutdownChattingServerThreadCreator.java

3. The Chatting Client

If you have understood the chatting system based on CSServer, you must learn that the chatting client has no any updates in its internal techniques since it primary tasks are sending notifications and sending requests. Both of them are presented in Chapter 3 and Chapter 4.

One tiny updates are related to the polling functions since the chatting client needs to send polling requests to the chatting server periodically. Compared with the client in Chapter 3 and Chapter 4, the polling requests should be sent under the control of a timer rather than by a user's manual operation.

For that, the code at the client side is straightforward. I do not think it is necessary to explain the code line by line except critical APIs or patterns. The same as the code on the server side, the client code works on GreatFree APIs and idioms as well. When you program yourself, those code must be good samples for you to follow.

3.1 StartChatClient: Starting the Chatting Client

List 5.41 shows the starting up program for the chatting client. It should be easy to understand. When programming it, the primary APIs you need to use are the ones of ChatReader, ChatEventer and Scheduler. They are not difficult. Simplify speaking, GreatFree does not provide distinctive supports on the traditional client besides the three APIs and the simple patterns for them.

```
1   package com.greatfree.chat.client;
2
3   import java.util.Scanner;
4
5   import com.greatfree.chat.ChatConfig;
6   import com.greatfree.chat.ClientMenu;
7   import com.greatfree.chat.MenuOptions;
8   import com.greatfree.concurrency.Scheduler;
9   import com.greatfree.util.ServerStatus;
10  import com.greatfree.util.TerminateSignal;
11
12  /*
13   * This is a client that interacts with the chatting server through the manner of requesting/responding
14   * and notifying. The chatting server can only respond to the client after it requests. 09/21/2014, Bing Li
15   */
16
17  // Created: 04/23/2017, Bing Li
18  public class StartChatClient
19  {
20      /*
21       * The starting point of the chatting client. 09/21/2014, Bing Li
22       */
23      public static void main(String[] args)
24      {
25          // Initialize the option which represents a user's intents of operations. 09/21/2014, Bing Li
26          int option = MenuOptions.NO_OPTION;
27
28          // Initialize a command input console for users to interact with the system. 09/21/2014, Bing Li
29          Scanner in = new Scanner(System.in);
30
31          // Input the local user name. 05/26/2017, Bing Li
32          System.out.println("Tell me your user name: ");
33
34          String username = in.nextLine();
35
36          // Input the chatting partner name. 05/26/2017, Bing Li
37          System.out.println("Tell me your partner: ");
38
39          String partner = in.nextLine();
40
41          // Initialize the status of the local node. 05/26/2017, Bing Li
42          ServerStatus.FREE().init();
43
44          // Initialize the client reader. 05/26/2017, Bing Li
45          ChatReader.RR().init();
46
47          // Initialize the scheduler to do something periodical. 02/02/2016, Bing Li
48          Scheduler.GREATFREE().init(ChatConfig.SCHEDULER_THREAD_POOL_SIZE,
49              ChatConfig.SCHEDULER_THREAD_POOL_KEEP_ALIVE_TIME);
50
51          // Initialize the chatting eventer. 04/28/2017, Bing Li
52          ChatEventer.RE().init();
53
54          // The chatting information maintainer. 05/25/2017, Bing Li
```

```
55          ChatMaintainer.CS().init(username, partner);
56
57          // Submit the polling tasks. 05/26/2017, Bing Li
58          Scheduler.GREATFREE().submit(new Checker(), ChatConfig.CHAT_POLLING_DELAY,
59              ChatConfig.CHAT_POLLING_PERIOD);
60
61          String optionStr;
62          // Keep the loop running to interact with users until an end option is selected. 09/21/2014, Bing Li
63          while (option != MenuOptions.QUIT)
64          {
65              // Display the menu to users. 09/21/2014, Bing Li
66              ClientUI.CS().printMenu();
67              // Input a string that represents users' intents. 09/21/2014, Bing Li
68              optionStr = in.nextLine();
69              try
70              {
71                  // Convert the input string to integer. 09/21/2014, Bing Li
72                  option = Integer.parseInt(optionStr);
73                  System.out.println("Your choice: " + option);
74
75                  // Send the option to the polling server. 09/21/2014, Bing Li
76                  ClientUI.CS().send(option);
77              }
78              catch (NumberFormatException e)
79              {
80                  option = MenuOptions.NO_OPTION;
81                  System.out.println(ClientMenu.WRONG_OPTION);
82              }
83          }
84
85          // Set the status of the local node as shutdown. 05/26/2017, Bing Li
86          ServerStatus.FREE().setShutdown();
87          // Set the terminating flag to true. 09/21/2014, Bing Li
88          TerminateSignal.SIGNAL().setTerminated();
89
90          // Dispose the chatting maintainer. 05/26/2017, Bing Li
91          ChatMaintainer.CS().dispose();
92
93          // Shutdown the scheduler. 02/02/2016, Bing Li
94          Scheduler.GREATFREE().shutdown();
95
96          try
97          {
98              // Dispose the chatting eventer. 05/25/2017, Bing Li
99              ChatEventer.RE().dispose();
100         }
101         catch (InterruptedException | IOException e )
102         {
103             ServerStatus.FREE().printException(e);
104         }
105
106         // Shutdown the remote reader. 11/23/2014, Bing Li
107         ChatReader.RR().shutdown();
108
109         in.close();
110     }
111 }
```

List 5.41 The code of StartChatClient.java

Line 45 initializes the chatting reader, which is used to send requests to the chatting server. The most important ones are the polling requests, including the one to check new chatting sessions and the one to check new chatting messages.

Line 48 initializes the scheduler which is responsible for sending the polling requests periodically.

Line 52 initializes the chatting eventer, which sends notifications to the chatting server. The two primary notifications in the case are the one that adds potential chatting partners and the one that sends new chatting messages.

Line 55 initializes the chatting maintainer, which does not use any GreatFree APIs and patterns but just an ordinary object implementation.

Line 58 submits the polling tasks to the scheduler such that polling requests can be sent periodically.

Line 86~109 collects all of the resources that the chatting client employs.

3.2 RR – The Pattern of Remote Reader: Sending Requests

The code of ChatReader simply invokes the API, RemoteReader, supported by GreatFree. When sending requests to a CSServer, the RemoteReader is the perfect choice for developers to achieve the goal. When programming the code, the pattern of the Remote Reader (RR) should be employed. List 5.42 is an example of the pattern, RR. Table 5.25 shows the summarization about the pattern of RR for the chatting client.

Pattern: RR (Remote Reader)		
Class	ChatReader	
Singleton	Yes	
Method	void init()	
	void shutdown()	
	ChatRegistryResponse registerChat(String userKey, String userName, String description)	
	ChatPartnerResponse searchUser(String userKey)	
	PollNewSessionsResponse checkNewSessions(String receiverKey, String username)	
	PollNewChatsResponse checkNewChats(String sessionKey, String receiverKey, String username)	
Enclosed Pattern	Invocation	RemoteReader.read(String nodeKey, String ip, int port, ServerMessage request)
Employed API	RemoteReader	

Table 5.25 The pattern of RR for the chatting client

The pattern of RR is very simple. The API, RemoteReader, is the primary component. In addition, it needs to include the request and the response before making it work.

```
1    package com.greatfree.chat.client;
2
3    import java.io.IOException;
4
5    import com.greatfree.chat.ChatConfig;
6    import com.greatfree.chat.message.ChatMessageConfig;
7    import com.greatfree.chat.message.cs.ChatPartnerRequest;
8    import com.greatfree.chat.message.cs.ChatPartnerResponse;
9    import com.greatfree.chat.message.cs.ChatRegistryRequest;
10   import com.greatfree.chat.message.cs.ChatRegistryResponse;
11   import com.greatfree.chat.message.cs.PollNewChatsRequest;
12   import com.greatfree.chat.message.cs.PollNewChatsResponse;
13   import com.greatfree.chat.message.cs.PollNewSessionsRequest;
14   import com.greatfree.chat.message.cs.PollNewSessionsResponse;
15   import com.greatfree.exceptions.RemoteReadException;
16   import com.greatfree.remote.RemoteReader;
17   import com.greatfree.testing.data.ClientConfig;
18   import com.greatfree.util.NodeID;
19   import com.greatfree.util.ServerStatus;
20
```

```
21  /*
22   * The class wraps the class, RemoteReader, to send requests to the remote server and wait until relevant
23   * responses are received. 04/23/2017, Bing Li
24   */
25
26  // Created: 04/23/2017, Bing Li
27  public class ChatReader
28  {
29      private ChatReader()
30      {
31      }
32
33      /*
34       * Initialize a singleton. 11/27/2014, Bing Li
35       */
36      private static ChatReader instance = new ChatReader();
37
38      public static ChatReader RR()
39      {
40          if (instance == null)
41          {
42              instance = new ChatReader();
43              return instance;
44          }
45          else
46          {
47              return instance;
48          }
49      }
50
51      /*
52       * Initialize the remote reader to send requests and receive responses from the remote
53       * server. 11/23/2014, Bing Li
54       */
55      public static void init()
56      {
57          RemoteReader.REMOTE().init(ClientConfig.CLIENT_READER_POOL_SIZE);
58      }
59
60      /*
61       * Shutdown the remote reader. 11/23/2014, Bing Li
62       */
63      public static void shutdown()
64      {
65          try
66          {
67              RemoteReader.REMOTE().shutdown();
68          }
69          catch (IOException e)
70          {
71              ServerStatus.FREE().printException(e);
72          }
73      }
74
75      /*
76       * Register the newly signed in user. 05/26/2017, Bing Li
77       */
78      public static ChatRegistryResponse registerChat(String userKey, String userName, String description)
79      {
80          try
81          {
82              return (ChatRegistryResponse)(RemoteReader.REMOTE().read
83                  (NodeID.DISTRIBUTED().getKey(), ChatConfig.CHAT_SERVER_ADDRESS,
84                  ChatConfig.CHAT_SERVER_PORT, new ChatRegistryRequest(userKey,
85                  userName, description)));
86          }
87          catch (ClassNotFoundException | RemoteReadException | IOException e)
88          {
89              e.printStackTrace();
90          }
91          return ChatMessageConfig.NO_CS_CHAT_REGISTRY_RESPONSE;
92      }
93
94      /*
95       * Search one particular chatting partner. 05/26/2017, Bing Li
96       */
97      public static ChatPartnerResponse searchUser(String userKey)
```

```
98          {
99              try
100             {
101                 return (ChatPartnerResponse)(RemoteReader.REMOTE().read(NodeID.DISTRIBUTED().getKey(),
102                     ChatConfig.CHAT_SERVER_ADDRESS, ChatConfig.CHAT_SERVER_PORT, new
103                     ChatPartnerRequest(userKey)));
104             }
105             catch (ClassNotFoundException | RemoteReadException | IOException e)
106             {
107                 e.printStackTrace();
108             }
109             return ChatMessageConfig.NO_CS_CHAT_PARTNER_RESPONSE;
110         }
111
112         /*
113          * Poll the chatting server to check whether new chatting sessions are available. 05/26/2017, Bing Li
114          */
115         public static PollNewSessionsResponse checkNewSessions(String receiverKey, String username)
116         {
117             try
118             {
119                 return (PollNewSessionsResponse)(RemoteReader.REMOTE().read
120                     (NodeID.DISTRIBUTED().getKey(), ChatConfig.CHAT_SERVER_ADDRESS,
121                     ChatConfig.CHAT_SERVER_PORT, new PollNewSessionsRequest
122                     (receiverKey, username)));
123             }
124             catch (ClassNotFoundException | RemoteReadException | IOException e)
125             {
126                 ServerStatus.FREE().setShutdown();
127                 ServerStatus.FREE().printException(e);
128             }
129             return ChatMessageConfig.NO_POLL_NEW_SESSIONS_RESPONSE;
130         }
131
132         /*
133          * Poll the chatting server to check whether new chatting messages are available. 05/26/2017, Bing Li
134          */
135         public static PollNewChatsResponse checkNewChats(String sessionKey, String receiverKey,
136             String username)
137         {
138             try
139             {
140                 return (PollNewChatsResponse)(RemoteReader.REMOTE().read
141                     (NodeID.DISTRIBUTED().getKey(), ChatConfig.CHAT_SERVER_ADDRESS,
142                     ChatConfig.CHAT_SERVER_PORT, new PollNewChatsRequest(sessionKey,
143                     receiverKey, username)));
144             }
145             catch (ClassNotFoundException | RemoteReadException | IOException e)
146             {
147                 ServerStatus.FREE().setShutdown();
148                 ServerStatus.FREE().printException(e);
149             }
150             return ChatMessageConfig.NO_POLL_NEW_CHATS_RESPONSE;
151         }
152     }
```

List 5.42 The code of ChatReader.java

Line 29~49 defines a singleton for the remote reader.

Line 55~58 initializes the API, RemoteReader. To manage the TCP clients for reading, the size of the client reader size needs to be specified.

Line 63~73 shuts down the RemoteReader.

Line 78~92 sends the user registering request to the chatting server.

Line 97~110 sends the request to the chatting server to search one potential chatting partner.

Line 115~130 sends the polling request to the chatting server to check whether new chatting sessions are available or not.

Line 135~151 sends the polling request to the chatting server to check whether new chatting messages are available or not.

3.3 RE – The Pattern of Remote Eventer: Sending Chatting Messages

Besides sending requests to the chatting server, the client also needs to send notifications. In the chatting application, the notifications include the one to add one potential partner (AddPartnerNotification) and the one to send new chatting messages (ChatNotification).

To program the code, you can follow the pattern of Remote Eventer (RE). Besides the APIs, AsyncRemoteEventer and SyncRemoteEventer, you need to define notification messages for both of them. Additionally, you are also required to define one thread pool (Line 22), which is used to execute instances of AsyncRemoteEventer, and one TCP client pool (Line 25) to send notifications efficiently.

Incidentally, when invoking the API of AsyncRemoteEventer, it needs to follow the pattern between Line 113~125 or Line 130~144. To simplify our description, the code is regarded as the pattern of ARE although it is not so critical to GreatFree. Table 5.26 summarizes the RE pattern for the chatting client.

Pattern: RE (Remote Eventer)		
Class	ChatEventer	
Singleton	Yes	
Method	void init()	
	void dispose()	
	void notifyAddFriend()	
	void notifyChat(String message)	
Enclosed Pattern	ARE (Asynchronous Remote Eventer)	AsyncRemoteEventer<AddPartnerNotification>
		AsyncRemoteEventer<ChatNotification>
Employed API	ThreadPool	
	FreeClientPool	
	AsyncRemoteEventer<Notification extends ServerMessage>	

Table 5.26 The pattern of RE for the chatting client

List 5.43 illustrates the specific example of the pattern. In this case, no synchronous eventers are defined because the feature is needed in the chatting system. Actually, you must not feel strange for the code. List 3.10 in Chapter 3 also gives you a more complete sample of RE since both SyncRemoteEventer and AsyncRemoteEventer are employed.

```
1    package com.greatfree.chat.client;
2
3    import com.greatfree.chat.ChatConfig;
4    import com.greatfree.chat.ChatTools;
5    import com.greatfree.chat.message.cs.AddPartnerNotification;
6    import com.greatfree.chat.message.cs.ChatNotification;
7    import com.greatfree.concurrency.ThreadPool;
8    import com.greatfree.remote.AsyncRemoteEventer;
```

```
9     import com.greatfree.testing.data.ClientConfig;
10
11    // Created: 04/23/2017, Bing Li
12    public class ChatEventer
13    {
14        // The notification of AddFriendNotification is sent to the chatting server in an asynchronous
15        // manner. 04/17/2017, Bing Li
16        private AsyncRemoteEventer<AddPartnerNotification> addFriendNotificationEventer;
17        // The notification of ChatNotification is sent to the chatting server in an asynchronous
18        // manner. 04/27/2017, Bing Li
19        private AsyncRemoteEventer<ChatNotification> chatNotificationEventer;
20
21        // A thread pool to assist sending notification asynchronously. 11/07/2014, Bing Li
22        private ThreadPool pool;
23
24        // The TCP client pool that sends notifications to the chatting server. 05/27/2017, Bing Li
25        private FreeClientPool clientPool;
26
24        /*
25         * Initialize. 11/07/2014, Bing Li
26         */
27        private ChatEventer()
28        {
29        }
30
31        /*
32         * A singleton implementation. 11/07/2014, Bing Li
33         */
34        private static ChatEventer instance = new ChatEventer();
35
36        public static ChatEventer RE()
37        {
38            if (instance == null)
39            {
40                instance = new ChatEventer();
41                return instance;
42            }
43            else
44            {
45                return instance;
46            }
47        }
48
49        public void dispose() throws InterruptedException
50        {
51            // Shutdown the client pool. 05/26/2017, Bing Li
52            this.clientPool.dispose();
53            // Shutdown the eventer that sends the notification of AddFriendNotification in an asynchronous
54            // manner. 11/27/2014, Bing Li
55            this.addFriendNotificationEventer.dispose();
56            // Shutdown the eventer that sends the notification of ChatNotification in an asynchronous
57            // manner. 11/27/2014, Bing Li
58            this.chatNotificationEventer.dispose();
59            // Shutdown the thread pool. 11/07/2014, Bing Li
60            this.pool.shutdown();
61        }
62
63        /*
64         * Initialize the eventers. 11/07/2014, Bing Li
65         */
66        public void init()
67        {
68             // Initialize the TCP client pool to send messages efficiently. 02/02/2016, Bing Li
69            this.clientPool = new FreeClientPool(ChatConfig.CLIENT_POOL_SIZE);
70            // Set idle checking for the client pool. 04/17/2017, Bing Li
71            this.clientPool.setIdleChecker(ChatConfig.CLIENT_IDLE_CHECK_DELAY,
72                ChatConfig.CLIENT_IDLE_CHECK_PERIOD, ChatConfig.CLIENT_MAX_IDLE_TIME);
73
74            this.pool = new ThreadPool(ClientConfig.EVENTER_THREAD_POOL_SIZE,
75                ClientConfig.EVENTER_THREAD_POOL_ALIVE_TIME);
76
77            // Initialize the asynchronous eventer for AddFriendNotification. 04/27/2017, Bing Li
78            this.addFriendNotificationEventer = new AsyncRemoteEventer.AsyncRemoteEventerBuilder
79                <AddPartnerNotification>()
80                    .threadPool(this.pool)
81                    .clientPool(this.clientPool)
82                    .eventQueueSize(ChatConfig.ASYNC_EVENT_QUEUE_SIZE)
```

```
83                  .eventerSize(ChatConfig.ASYNC_EVENTER_SIZE)
84                  .eventingWaitTime(ChatConfig.ASYNC_EVENTING_WAIT_TIME)
85                  .eventerWaitTime(ChatConfig.ASYNC_EVENTER_WAIT_TIME)
86                  .waitRound(ChatConfig.ASYNC_EVENTER_WAIT_ROUND)
87                  .idleCheckDelay(ChatConfig.ASYNC_EVENT_IDLE_CHECK_DELAY)
88                  .idleCheckPeriod(ChatConfig.ASYNC_EVENT_IDLE_CHECK_PERIOD)
89                  .schedulerPoolSize(ClientConfig.SCHEDULER_POOL_SIZE)
90                  .schedulerKeepAliveTime(ClientConfig.SCHEDULER_KEEP_ALIVE_TIME)
91                  .build();
92
93          // Initialize the asynchronous eventer for ChatNotification. 04/27/2017, Bing Li
94          this.chatNotificationEventer = new AsyncRemoteEventer.AsyncRemoteEventerBuilder
95              <ChatNotification>()
96                  .threadPool(this.pool)
97                  .clientPool(this.clientPool)
98                  .eventQueueSize(ChatConfig.ASYNC_EVENT_QUEUE_SIZE)
99                  .eventerSize(ChatConfig.ASYNC_EVENTER_SIZE)
100                 .eventingWaitTime(ChatConfig.ASYNC_EVENTING_WAIT_TIME)
101                 .eventerWaitTime(ChatConfig.ASYNC_EVENTER_WAIT_TIME)
102                 .waitRound(ChatConfig.ASYNC_EVENTER_WAIT_ROUND)
103                 .idleCheckDelay(ChatConfig.ASYNC_EVENT_IDLE_CHECK_DELAY)
104                 .idleCheckPeriod(ChatConfig.ASYNC_EVENT_IDLE_CHECK_PERIOD)
105                 .schedulerPoolSize(ClientConfig.SCHEDULER_POOL_SIZE)
106                 .schedulerKeepAliveTime(ClientConfig.SCHEDULER_KEEP_ALIVE_TIME)
107                 .build();
108     }
109
110     /*
111      * Notify the chatting server to shut down the chatting server. 04/17/2017, Bing Li
112      */
113     public void notifyAddFriend()
114     {
115         // Check whether the eventer manager is ready to send asynchronous notifications. 04/17/2017, Bing Li
116         if (!this.addFriendNotificationEventer.isReady())
117         {
118             // Execute the eventer manager. 04/17/2017, Bing Li
119             this.pool.execute(this.addFriendNotificationEventer);
120         }
121         // Send the notification. 04/17/2017, Bing Li
122         this.addFriendNotificationEventer.notify(ChatConfig.CHAT_SERVER_ADDRESS,
123             ChatConfig.CHAT_SERVER_PORT, new AddPartnerNotification
124             (ChatMaintainer.CS().getLocalUserKey(), ChatMaintainer.CS().getPartnerKey(), "Hello!"));
125     }
126
127     /*
128      * Notify the chatting server to shut down the chatting server. 04/17/2017, Bing Li
129      */
130     public void notifyChat(String message)
131     {
132         // Check whether the eventer manager is ready to send asynchronous notifications. 04/17/2017, Bing Li
133         if (!this.chatNotificationEventer.isReady())
134         {
135             // Execute the eventer manager. 04/17/2017, Bing Li
136             this.pool.execute(this.chatNotificationEventer);
137         }
138         // Send the notification. 04/17/2017, Bing Li
139         this.chatNotificationEventer.notify(ChatConfig.CHAT_SERVER_ADDRESS,
140             ChatConfig.CHAT_SERVER_PORT, new ChatNotification(ChatTools.getChatSessionKey
141             (ChatMaintainer.CS().getLocalUserKey(), ChatMaintainer.CS().getPartnerKey()), message,
142             ChatMaintainer.CS().getLocalUserKey(), ChatMaintainer.CS().getLocalUsername(),
143             ChatMaintainer.CS().getPartnerKey(), ChatMaintainer.CS().getPartner()));
144     }
145 }
```

List 5.43 The code of ChatEventer.java

Similar to ChatReader, the ChatEventer is also implemented as a singleton. Two GreatFree APIs are utilized in the code. The first one is AsyncRemoteEventer and the second one is ThreadPool. The API of AsyncRemoteEventer sends notifications to the chatting server in a non-blocking manner. When no synchronization is required in a specific scenario, it is recommended to use the API. Otherwise, you have to choose

the one, SyncRemoteEventer, which sends notifications in a blocking manner. The thread pool is used to execute the asynchronous eventer since the eventer itself is a thread as well.

Line 16~25 declares the two asynchronous eventers to send the two notifications, AddPartnerNotification and ChatNotification.

Line 22 declares the instance of ThreadPool to execute the asynchronous eventer to send notification in a non-blocking manner.

Line 49~61 disposes all of the eventing resources consumed in the chatting client.

Line 66~108 initializes all of the eventing resources to send notifications. You must find that a bunch of parameters should be set when doing those initializations.

Line 69~71 initializes the TCP client pool, an instance of FreeClientPool supported by GreatFree. It manages the instances of TCP client. Those clients are used to send notifications to the chatting server.

To initialize the TCP client pool, it is necessary to specific the size of the pool, ChatConfig.CLINET_POOL_SIZE, as done in Line 69. In addition, since the client pool needs to check whether one client is idle long enough such that it is required to specify the delay, ChatConfig.CLIENT_IDLE_CHECK_DELAY, and the period, ChatConfig.CLIENT_IDLE_CHECK_PERIOD. When one TCP client is idle long enough, it should be collected. It also need to set the length of the idle time, ChatConfig.CLIENT_MAX_IDLE_TIME. All of the sets are done in Line 71.

Line 74 initializes the thread pool that is used to execute the instance of AsyncRemoteEventer. When initializing the thread pool, you have to set the size of the pool, ClientConfig.EVENTER_THREAD_POOL_SIZE and the time to keep those idle threads alive, ClientConfig.EVENTER_THREAD_POOL_ALIVE_TIME.

To initialize the asynchronous eventer, more parameters are required, as listed in Table 5.27.

Line 78~91 initializes the instance, addFriendNotificationEventer, of the class, AsyncRemoteEventer. All of the parameters of its constructor are discussed in Table 5.27.

Line 94~107 initializes the instance, chatNotificationEventer, of the class, AsyncRemoteEventer. You can also follow Table 5.27 to set up the parameters of its constructor.

Line 113~125 defines the method, notifyAddFriend(), to send the asynchronous notification of AddPartnerNotification to the chatting server. To do that, it is required to check whether the eventer is ready or not, which is done in Line 116. If not ready, it can be executed by the thread pool to get it done, as shown in Line 119. Line 122 enqueues the notification to the eventer for sending asynchronously. You can see the structure here is identical to those in the patterns of SD and RD as presented in Section 2.5 and Section 2.6.

Line 130~144 defines the method, notifyChat(), to send the asynchronous notification of ChatNotification to the chatting server. The structure of the method is the same as that for AddPartnerNotification.

Parameter	Explanation
ThreadPool pool	The thread pool that executes the asynchronous eventer to send notification in a non-blocking manner
FreeClientPool clientPool	The TCP client pool that manages the instances of TCP clients to interact with the chatting server
int eventQueueSize	The length of the built-in queue of Eventer that takes notifications sequentially to be sent asynchronously. It is necessary to emphasize that AsyncRemoteEventer is also a thread pool to manage instances of Eventer, which the thread derived from NotificationObjectQueue, which is similar to the one of NotificationQueue as discussed in Section 2.6.1 and 2.6.2.
int eventerSize	The maximum size of Eventer instances that can be created in the asynchronous eventer, AsyncRemoteEventer
long eventingWaitTime	When no notifications are available, the instance of AsyncRemoteEventer needs to wait for some time before disposing itself to save resources. The time is called, eventingWaitTime
long eventerWaitTime	The parameter is easily confused with the one of eventingWaitTime. AsyncRemoteEventer is a pool to manage instances of Eventer, which is actually an instance of NotificationObjectQueue. The thread needs to wait for some time when its queue is empty. That time is called, eventerWaitTime
int waitRound	When no notifications should be processed, to ensure whether it is time to dispose itself, it is necessary to wait for a couple of times before doing that. One time is identical to the round. It indicates that the instance of AsyncRemoteEventer has waited for a couple of periods, each of which is equal to eventingWaitTime. If the value is higher, it represents that it should wait for more rounds, and vice versa. The similar parameters emerge in Table 5.5 and Table 5.6
long idleCheckDelay	The AsyncRemoteEventer needs to check whether alive instances of Eventer is idle long enough. The value is the delay before the periodical checking is started
long idleCheckPeriod	The length of the period to check whether alive instances of Eventer are idle long enough
int schedulerPoolSize	The size of the thread pool inside the scheduler, which is used to check periodically whether eventers in the AysncRemoteEventer is idle or not. If idle, they should be collected
long schedulerKeepAliveTime	The keeping alive time for thread pool inside the scheduler

Table 5.27 The parameters of the constructor of AsyncRemoteEventer

3.4 The Application Code

Until now, all of the patterns for the chatting client, i.e., a traditional client of C/S model, are introduced. Unfortunately, your coding tasks have not been done. You have to add some application code to make your system work as what you do on the server side, such as AccountRegistry (List 4.3), CSAccount (List 5.4), PrivateChatSessions (List 5.5), ChatSession (List 5.6), ChatMessage (List 5.7), ChatTools (List 5.8) and so forth.

3.4.1 ChatMaintainer: Maintaining the Chatting Information

List 5.44 shows the code of ChatMaintainer. It keeps information about the chatting. I think it is easy to understand.

It is necessary to mention that the polling requests are sent from the ChatMaintainer. It invokes the reading methods of RR in Line 98~99 and Line 118~119. The relevant methods, checkNewSessions() (Line 96) and checkNewChats() (Line 113), of ChatMaintainer are invoked by the scheduler periodically.

```
1    package com.greatfree.chat.client;
2
3    import java.util.Set;
4
5    import com.google.common.collect.Sets;
6    import com.greatfree.chat.ChatMessage;
7    import com.greatfree.chat.ChatTools;
8    import com.greatfree.chat.message.cs.PollNewChatsResponse;
9    import com.greatfree.chat.message.cs.PollNewSessionsResponse;
10
11   /*
12   * The class checks whether new sessions or new chat messages are available on the
13   * chatting server. 04/24/2017, Bing Li
14    */
15
16   // Created: 04/24/2017, Bing Li
17   public class ChatMaintainer
18   {
19       private String localUserKey;
20       private String localUsername;
21       private String partner;
22       private String partnerKey;
23       private Set<String> participatedSessions;
24
25       private ChatMaintainer()
26       {
27           this.participatedSessions = Sets.newHashSet();
28       }
29
30       /*
31        * A singleton definition. 04/17/2017, Bing Li
32        */
33       private static ChatMaintainer instance = new ChatMaintainer();
34
35       public static ChatMaintainer CS()
36       {
37           if (instance == null)
38           {
39               instance = new ChatMaintainer();
40               return instance;
41           }
42           else
43           {
44               return instance;
45           }
46       }
47
48       /*
49        * Dispose the free client pool. 04/17/2017, Bing Li
50        */
51       public void dispose()
52       {
53           this.participatedSessions.clear();
54           this.participatedSessions = null;
55       }
56
57       public void init(String username, String partner)
58       {
59           this.localUsername = username;
60           this.localUserKey = ChatTools.getUserKey(username);
61           this.partner = partner;
62           this.partnerKey = ChatTools.getUserKey(partner);
63       }
64
65       public String getLocalUserKey()
66       {
67           return this.localUserKey;
68       }
69
70       public String getLocalUsername()
71       {
72           return this.localUsername;
73       }
74
75       public String getPartner()
```

```
76          {
77              return this.partner;
78          }
79
80          public String getPartnerKey()
81          {
82              return this.partnerKey;
83          }
84
85          /*
86           * Add the participated chat session. 04/24/2017, Bing Li
87           */
88          public void addSession(String sessionKey)
89          {
90              this.participatedSessions.add(sessionKey);
91          }
92
93          /*
94           * Check whether new sessions are available on the chatting server. 04/24/2017, Bing Li
95           */
96          public void checkNewSessions()
97          {
98              PollNewSessionsResponse response = ChatReader.RR().checkNewSessions(this.localUserKey,
99                  this.localUsername);
100             if (response.getNewSessionKeys() != null)
101             {
102                 System.out.println(response.getNewSessionKeys().size() + " new chatting
103                     sessions are obtained ...");
104                 this.participatedSessions.addAll(response.getNewSessionKeys());
105                 System.out.println(response.getNewSessionKeys().size() + " new chatting sessions are
106                     retained ...");
107             }
108         }
109
110         /*
111              * Check whether new chatting messages are available on the chatting server. 04/24/2017, Bing Li
112          */
113         public void checkNewChats()
114         {
115             PollNewChatsResponse response;
116             for (String sessionKey : this.participatedSessions)
117             {
118                 response = ChatReader.RR().checkNewChats(sessionKey, this.localUserKey,
119                     this.localUsername);
120                 if (response.getMessages().size() > 0)
121                 {
122                     System.out.println("PollNewChatsResponse: " + response.getMessages().size()
123                         + " messages received");
124                     for (ChatMessage message : response.getMessages())
125                     {
126                         System.out.println(message.getSenderName() + " says: " + message.getMessage());
127                     }
128                 }
129             }
130         }
131     }
```

List 5.44 The code of ChatMaintainer.java

3.4.2 Checker: The Polling Tasks

List 5.45 presents the code of Checker.java, which includes the tasks to send polling requests to the chatting server. It is executed by the Scheduler initialized in Line 48 of List 5.41 (StartChatClient.java). The task is submitted in Line 48 of the same code.

```
1       package com.greatfree.chat.client;
2
3       /*
4        * The class periodically checks whether new sessions and new messages are available on the
5        * chatting server. 04/24/2017, Bing Li
```

```
6       */
7
8       // Created: 04/24/2017, Bing Li
9       public class Checker implements Runnable
10      {
11          @Override
12          public void run()
13          {
14              ChatMaintainer.CS().checkNewSessions();
15              ChatMaintainer.CS().checkNewChats();
16          }
17      }
```

List 5.45 The code of Checker.java

3.4.3 ClientUI: Displaying a Simple Interface

List 5.46 shows the simple interface on the screen such that users can manipulate the chatting system and interact with the remote partner.

```
1       package com.greatfree.chat.client;
2
3       import java.util.Scanner;
4
5       import com.greatfree.chat.ChatOptions;
6       import com.greatfree.chat.ClientMenu;
7       import com.greatfree.chat.MenuOptions;
8       import com.greatfree.chat.message.cs.ChatPartnerResponse;
9       import com.greatfree.chat.message.cs.ChatRegistryResponse;
10
11      /*
12       * The class aims to print a menu list on the screen for users to interact with the client and communicate with
13       * the chatting server. The menu is unique in the client such that it is implemented in the pattern of a
14       * singleton. 04/23/2017, Bing Li
15       */
16
17      // Created: 04/23/2017, Bing Li
18      public class ClientUI
19      {
20          private Scanner in = new Scanner(System.in);
21
22          /*
23           * Initialize. 04/23/2017, Bing Li
24           */
25          private ClientUI()
26          {
27          }
28
29          /*
30           * Initialize a singleton. 04/23/2017, Bing Li
31           */
32          private static ClientUI instance = new ClientUI();
33
34          public static ClientUI CS()
35          {
36              if (instance == null)
37              {
38                  instance = new ClientUI();
39                  return instance;
40              }
41              else
42              {
43                  return instance;
44              }
45          }
46
47          public void dispose()
48          {
49              this.in.close();
50          }
```

```
51
52          /*
53           * Print the menu list on the screen. 04/23/2017, Bing Li
54           */
55          public void printMenu()
56          {
57              System.out.println(ClientMenu.MENU_HEAD);
58              System.out.println(ClientMenu.REGISTER_CHATTING + ChatMaintainer
59                  .CS().getLocalUsername());
60              System.out.println(ClientMenu.SEARCH_USER + ChatMaintainer.CS().getPartner());
61              System.out.println(ClientMenu.ADD_FRIEND + ChatMaintainer.CS().getPartner());
62              System.out.println(ClientMenu.START_CHATTING + ChatMaintainer.CS().getPartner());
63              System.out.println(ClientMenu.QUIT);
64              System.out.println(ClientMenu.MENU_TAIL);
65              System.out.println(ClientMenu.INPUT_PROMPT);
66          }
67
68          /*
69           * Send the users' option to the chatting server. 04/23/2017, Bing Li
70           */
71          public void send(int option)
72          {
73              ChatRegistryResponse chatRegistryResponse;
74              ChatPartnerResponse chatPartnerResponse;
75
76              // Check the option to interact with the chatting server. 04/23/2017, Bing Li
77              switch (option)
78              {
79                  case MenuOptions.REGISTER_CHATTING:
80                      chatRegistryResponse = ChatReader.RR().registerChat(ChatMaintainer.CS()
81                          .getLocalUserKey(), ChatMaintainer.CS().getLocalUsername(),
82                          ChatMaintainer.CS().getLocalUsername() + " is a great & free guy!");
83                      System.out.println("Chatting registry status: " + chatRegistryResponse.isSucceeded());
84                  break;
85
86                  case MenuOptions.SEARCH_USER:
87                      chatPartnerResponse = ChatReader.RR().searchUser(ChatMaintainer.CS().getPartnerKey());
88                      System.out.println(chatPartnerResponse.getUserName() + ": " +
89                          chatPartnerResponse.getDescription());
90                  break;
91
92                  case MenuOptions.ADD_FRIEND:
93                      ChatEventer.RE().notifyAddFriend();
94                  break;
95
96                  case MenuOptions.START_CHATTING:
97                      int chatOption = ChatOptions.NO_OPTION;
98                      String optionStr;
99                      while (chatOption != ChatOptions.QUIT_CHAT)
100                     {
101                         ChatUI.CS().printMenu();
102                         optionStr = in.nextLine();
103                         try
104                         {
105                             chatOption = Integer.parseInt(optionStr);
106                             System.out.println("Your choice: " + option);
107                             ChatUI.CS().sent(chatOption);
108                         }
109                         catch (NumberFormatException e)
110                         {
111                             chatOption = MenuOptions.NO_OPTION;
112                             System.out.println(ClientMenu.WRONG_OPTION);
113                         }
114                     }
115                 break;
116
117                 case MenuOptions.QUIT:
118                 break;
119             }
120         }
121     }
```

List 5.46 The code of ClientUI.java

You can find that ClientMenu is a new class in the method of send(). The class encloses some String constants to define the menu on the screen. The code is shown in List 5.47 as follows.

```
1    package com.greatfree.chat;
2
3    /*
4     * This is a simple menu for the client which is operated by users. 04/23/2017, Bing Li
5     */
6
7    // Created: 04/23/2017, Bing Li
8    public class ClientMenu
9    {
10       public final static String TAB = "   ";
11       public final static String MENU_HEAD = "\n========== Menu Head ===========";
12       public final static String REGISTER_CHATTING = ClientMenu.TAB + "1) Register Chatting: ";
13       public final static String SEARCH_USER = ClientMenu.TAB + "2) Search User: ";
14       public final static String ADD_FRIEND = ClientMenu.TAB + "3) Add Friend: ";
15       public final static String START_CHATTING = ClientMenu.TAB + "4) Start Chatting with: ";
16       public final static String QUIT = ClientMenu.TAB + "0) Quit";
17       public final static String MENU_TAIL = "========== Menu Tail ===========\n";
18       public final static String INPUT_PROMPT = "Input an option:";
19       public final static String WRONG_OPTION = "Wrong option!";
20   }
```

List 5.47 The code of ClientMenu.java

The method of send() in ClientUI has a switch-case statement. Its options are defined in the code of MenuOptions.java, as shown in List 5.48.

```
1    package com.greatfree.chat;
2
3    /*
4     * The class contains all of constants of menu options. 04/23/2017, Bing Li
5     */
6
7    // Created: 04/23/2017, Bing Li
8    public class MenuOptions
9    {
10       public final static int NO_OPTION = -1;
11       public final static int REGISTER_CHATTING = 1;
12       public final static int SEARCH_USER = 2;
13       public final static int ADD_FRIEND = 3;
14       public final static int START_CHATTING = 4;
15       public final static int QUIT = 0;
16   }
```

List 5.48 The code of MenuOptions.java

When a chatting is started, a sub menu is displayed on the screen. It is implemented by the code of ChatUI.java, as shown in List 5.49.

```
1    package com.greatfree.chat.client;
2
3    import java.util.Scanner;
4
5    import com.greatfree.chat.ChatMenu;
6    import com.greatfree.chat.ChatOptions;
7
8    /*
9     * The class aims to print a menu list on the screen for users to interact with the client and chat with the
10    * chatting server. The menu is unique in the client such that it is implemented in the pattern of
11    * a singleton. 04/27/2017, Bing Li
12    */
13
14   // Created: 04/27/2017, Bing Li
```

```
15    public class ChatUI
16    {
17        private Scanner in = new Scanner(System.in);
18
19        /*
20         * Initialize. 04/27/2017, Bing Li
21         */
22        private ChatUI()
23        {
24        }
25
26        /*
27         * Initialize a singleton. 04/23/2017, Bing Li
28         */
29        private static ChatUI instance = new ChatUI();
30
31        public static ChatUI CS()
32        {
33            if (instance == null)
34            {
35                instance = new ChatUI();
36                return instance;
37            }
38            else
39            {
40                return instance;
41            }
42        }
43
44        public void dispose()
45        {
46            this.in.close();
47        }
48
49        /*
50         * Print the menu list on the screen. 04/23/2017, Bing Li
51         */
52        public void printMenu()
53        {
54            System.out.println(ChatMenu.MENU_HEAD);
55            System.out.println(ChatMenu.TYPE_MESSAGE + ChatMaintainer.CS().getPartner());
56            System.out.println(ChatMenu.QUIT);
57            System.out.println(ChatMenu.MENU_TAIL);
58            System.out.println(ChatMenu.INPUT_PROMPT);
59        }
60
61        /*
62         * Send messages to the chatting server. 04/23/2017, Bing Li
63         */
64        public void sent(int option)
65        {
66            switch (option)
67            {
68                case ChatOptions.TYPE_CHAT:
69                    System.out.println("Please type your message: ");
70                    String message = in.nextLine();
71                    ChatEventer.RE().notifyChat(message);
72                break;
73
74                case ChatOptions.QUIT_CHAT:
75                break;
76            }
77        }
78    }
```

List 5.49 The code of ChatUI.java

To print the menu, the class, ChatMenu, should be defined. The code is shown in List 5.50.

```
1     package com.greatfree.chat;
2
```

```
3       /*
4        * This is a simple chatting menu for the client which is operated by users. 04/23/2017, Bing Li
5        */
6
7       // Created: 04/27/2017, Bing Li
8       public class ChatMenu
9       {
10          public final static String TAB = "   ";
11          public final static String MENU_HEAD = "\n========== Menu Head ==========";
12          public final static String TYPE_MESSAGE = ClientMenu.TAB + "1) Type Message To: ";
13          public final static String QUIT = ClientMenu.TAB + "0) Quit";
14          public final static String MENU_TAIL = "========== Menu Tail ==========\n";
15          public final static String INPUT_PROMPT = "Input an option:";
16          public final static String WRONG_OPTION = "Wrong option!";
17      }
```

List 5.50 The code of ChatMenu.java

The last piece of code, ChatOptions.java, defines the option for the chatting client. Its code is shown in List 5.51.

```
1       package com.greatfree.chat;
2
3       /*
4        * The class contains all of constants of chatting options. 04/27/2017, Bing Li
5        */
6
7       // Created: 04/27/2017, Bing Li
8       public class ChatOptions
9       {
10          public final static int NO_OPTION = -1;
11          public final static int TYPE_CHAT = 1;
12          public final static int QUIT_CHAT = 0;
13      }
```

List 5.51 The code of ChatOptions.java

4. The Chatting Administration

Until this section, the code of the chatting server and that of the chatting client are explained in detail. However, to construct a distributed system that is closed to be practical, it is not complete. The most apparent question is how to shut down the system. It seems to be an easy one. However, if you cannot manage computing resources in your system well, you cannot terminate your applications properly. For example, when a thread cannot be maintained, your system keeps running even though you send a command to shut down.

You might say any systems can be shut down by pressing the compound key, CTRL+C. I highly suggest you never do that again, especially for a practical system. Your system has to be shut down by a command rather than any system-level approaching, like CTRL+C or the one of kill.

Fortunately, when programming with GreatFree, all of resources raised by those APIs and idioms are managed by GreatFree for sure. If your application code can be collected properly, you do not need to worry about GreatFree resources.

In our chatting system, the chatting client can be shut down by choosing the menu of quitting. So you do not need to consider the client. However, since no menu is available on the chatting server, you have to choose another way. Luckily, you might remember the chatting server implements two patterns of SD, i.e., the one of ChatServerDispatcher (List 5.9) and the one of ChatManServerDispatcher (List 5.10). The first one is designed for receiving all of the messages related to the chatting application. In contrast, the later one is prepared for the notification of ShutdownChatServerNotification (List 5.38) only. If the chatting server receives the notification, it shuts down the server in the thread of the ND for the notification.

However, the notification of ShutdownChatServerNotification is not sent by the chatting client since it is not reasonable. Instead, an administrator is responsible for doing that. So now we need to take a look at the code of the administrator. Before that, you can imagine what the code will look like. How to implement it? Since the administrator needs to send the notification, it should be programmed like the traditional client, the chatting client, which we just show you, right? Yes, this is just the client. All of those APIs and idioms are repeated again. So, let us start.

4.1 Starting the Administrator

List 5.55 shows the code of the administrator, Administrator.java. Although we do not explain it as a pattern, it looks very similar to the one, StartChatClient.java, in List 5.41. You just pay a little bit attention to Line 25, which initializes the RE of ChatAdminEventer, and Line 29, which initializes the RR of ChatAdminReader.

```
1    package com.greatfree.chat.admin;
2
3    import java.io.IOException;
4    import java.util.Scanner;
5
6    import com.greatfree.admin.AdminConfig;
7    import com.greatfree.admin.Menu;
8    import com.greatfree.chat.ChatConfig;
9
10   /*
11    * This is the administrator to manage the chatting system. 04/17/2016, Bing Li
12    */
13
14   // Created: 04/17/2017, Bing Li
15   public class Administrator
16   {
17       public static void main(String[] args)
18       {
19           // Initialize the option, which represents the commands. 11/27/2014, Bing Li
20           int option = AdminConfig.NO_OPTION;
21           // Initialize a scanner to wait for the administrator's commands. 11/27/2014, Bing Li
22           Scanner in = new Scanner(System.in);
23
24           // Initialize the administrator eventer. 05/27/2017, Bing Li
25           ChatAdminEventer.RE().init();
26
27           // Initialize the remote reader to send requests and receive responses from the register server. Usually,
28           // it needs to get IPs of the peer based servers. 11/23/2014, Bing Li
29           ChatAdminReader.RR().init();
30
31           String optionStr;
32           while (option != AdminConfig.END)
33           {
34               // Print the console menu. 04/17/2017, Bing Li
35               printMenu();
36               // Wait for the administrator's input from the console. 11/27/2014, Bing Li
```

```
37                  optionStr = in.nextLine();
38                  try
39                  {
40                      // Convert the string to the integer. 11/27/2014, Bing Li
41                      option = Integer.parseInt(optionStr);
42                      // Print the option. 11/27/2014, Bing Li
43                      System.out.println("Your choice is: " + option);
44                      // Notify the coordinator. 11/27/2014, Bing Li
45                      notifyServer(option);
46                  }
47                  catch (NumberFormatException e)
48                  {
49                      option = AdminConfig.NO_OPTION;
50                      System.out.println(Menu.WRONG_OPTION);
51                  }
52              }
53
54              try
55              {
56                  // Dispose the eventer. 11/27/2014, Bing Li
57                  ChatAdminEventer.RE().dispose();
58              }
59              catch (InterruptedException | IOException e)
60              {
61                  e.printStackTrace();
62              }
63
64              // Shutdown the remote reader. 11/23/2014, Bing Li
65              ChatAdminReader.RR().shutdown();
66
67              in.close();
68          }
69
70          /*
71           * Print the console menu for the administrator. 11/27/2014, Bing Li
72           */
73          private static void printMenu()
74          {
75              System.out.println(Menu.MENU_HEAD);
76              System.out.println(ChatMenu.STOP_CHATTING_SERVER);
77              System.out.println(ChatMenu.STOP_CHATTING_REGISTRY_SERVER);
78              System.out.println(ChatMenu.STOP_CLUSTER_CHILDREN);
79              System.out.println(Menu.END);
80              System.out.println(Menu.MENU_TAIL);
81              System.out.println(Menu.INPUT_PROMPT);
82          }
83
84          /*
85           * Notify the chatting server to manage the distributed system. 11/27/2014, Bing Li
86           */
87          private static void notifyServer(int option)
88          {
89              // Check the option of the administrator. 11/27/2014, Bing Li
90              switch (option)
91              {
92                  // Shutdown the chatting server. 11/27/2014, Bing Li
93                  case ChatConfig.STOP_CS_CHATTING_SERVER:
94                      System.out.println(ChatMenu.STOP_CHATTING_SERVER);
95                      // Notify the CS chatting server to shutdown. 11/27/2014, Bing Li
96                      ChatAdminEventer.RE().notifyShutdownCSChatServer();
97                      break;
98
99                  // Shutdown the chatting registry server. 05/01/2017, Bing Li
100                 case ChatConfig.STOP_CHATTING_REGISTRY_SERVER:
101                     System.out.println(ChatMenu.STOP_CHATTING_REGISTRY_SERVER);
102                     // Notify the chatting registry server to shutdown. 11/27/2014, Bing Li
103                     ChatAdminEventer.RE().notifyShutdownChattingRegistryServer();
104                     break;
105
106                 // Shutdown the cluster server. 05/01/2017, Bing Li
107                 case ChatConfig.STOP_CLUSTER_CHILDREN:
108                     System.out.println(ChatMenu.STOP_CLUSTER_CHILDREN);
109                     // Notify the cluster root to shutdown children. 11/27/2014, Bing Li
110                     ChatAdminEventer.RE().notifyShutdownClusterChildren();
111                     break;
112             }
113         }
```

```
114 }
```

List 5.52 The code of Administrator.java

4.2 RE: The Eventer of the Administrator

As a traditional client, it usually has two important pattern, the RR (Remote Reader) and the RE (Remote Eventer), as we discuss previously. In the case of the chatting administrator, if it just need to send remote management notifications to the chatting server only, it is unnecessary to have the RR. For the chatting system implemented by the distributed model of C/S, that is true. However, if we do that with other models, it must be changed. We will explain that in the later chapter. Table 5.28 lists the pattern of RE for the administrator.

List 5.56 presents the code of the eventer of the chatting administrator. Compared with the eventer of the chatting client in List 5.43, it is also an application of the RE pattern. Both of them has the exactly same structure, such as the asynchronous eventer, the thread pool, the TCP client pool, the singleton structure and so forth.

The RE needs to send two notifications, ShutdownChatServerNotification and ShutdownChildrenAdminNotification. The first one is the one we need to shut down our chatting server whereas the second one is not useful in the case of the chapter.

Pattern: RE (Remote Eventer)		
Class	ChatAdminEventer	
Singleton	Yes	
Method	void init()	
	void dispose()	
	void notifyShutdownChattingPeer()	
	void notifyShutdownClusterChildren()	
Enclosed Pattern	ARE (Asynchronous Remote Eventer)	AsyncRemoteEventer<ShutdownChatServerNotification>
		AsyncRemoteEventer<ShutdownChildrenAdminNotification>
Employed API	ThreadPool	
	FreeClientPool	
	AsyncRemoteEventer<Notification extends ServerMessage>	

Table 5.28 The pattern of RE for the administrator

```
1   package com.greatfree.chat.admin;
2
3   import java.io.IOException;
4
5   import com.greatfree.chat.ChatConfig;
6   import com.greatfree.chat.message.ShutdownChatServerNotification;
7   import com.greatfree.chat.message.peer.ChatPartnerResponse;
8   import com.greatfree.cluster.ClusterConfig;
9   import com.greatfree.cluster.message.ShutdownChildrenAdminNotification;
10  import com.greatfree.concurrency.ThreadPool;
11  import com.greatfree.remote.AsyncRemoteEventer;
12  import com.greatfree.remote.FreeClientPool;
13  import com.greatfree.remote.RemoteReader;
14  import com.greatfree.testing.data.ClientConfig;
15
16  /*
17   * This is an eventer that sends notifications to the chatting server for administration in a synchronous or
18   * asynchronous manner. 04/17/2014, Bing Li
19   */
20
21  // Created: 04/17/2017, Bing Li
22  public class ChatAdminEventer
```

```
23  {
24      // The notification of ShutdownChatServerNotification is sent to the chatting server in an
25      // asynchronous manner. 04/17/2017, Bing Li
26      private AsyncRemoteEventer<ShutdownChatServerNotification>
27          shutdownChattingServerNotificationEventer;
28      // The notification of ShutdownChildrenAdminNotification is sent to the cluster root in an
29      // asynchronous manner. 04/17/2017, Bing Li
30      private AsyncRemoteEventer<ShutdownChildrenAdminNotification>
31          shutdownClusterChildrenNotificationEventer;
32
33      // The thread pool that starts up the asynchronous eventer. 04/17/2017, Bing Li
34      private ThreadPool pool;
35
36      // The TCP client pool that sends notifications to the chatting server. 05/27/2017, Bing Li
37      private FreeClientPool clientPool;
38
39      private ChatAdminEventer()
40      {
41      }
42
43      /*
44       * Initialize a singleton. 11/27/2014, Bing Li
45       */
46      private static ChatAdminEventer instance = new ChatAdminEventer();
47
48      public static ChatAdminEventer RE()
49      {
50          if (instance == null)
51          {
52              instance = new ChatAdminEventer();
53              return instance;
54          }
55          else
56          {
57              return instance;
58          }
59      }
60
61      /*
62       * Dispose the eventer. 11/27/2014, Bing Li
63       */
64      public void dispose() throws InterruptedException, IOException
65      {
66          // Shutdown the client pool. 05/26/2017, Bing Li
67          this.clientPool.dispose();
68          // Shutdown the eventer that sends the notification of ShutdownChatServerNotification to the CS
69          // chatting server in an asynchronous manner. 11/27/2014, Bing Li
70          this.shutdownChattingServerNotificationEventer.dispose();
71          // Shutdown the eventer that sends the notification of ShutdownChildrenAdminNotification to the cluster
72          // root in an asynchronous manner. 11/27/2014, Bing Li
73          this.shutdownClusterChildrenNotificationEventer.dispose();
74          // Shutdown the thread pool. 11/27/2014, Bing Li
75          this.pool.shutdown();
76      }
77
78      /*
79       * Initialize the eventer. 11/27/2014, Bing Li
80       */
81      public void init()
82      {
83           // Initialize the TCP client pool to send messages efficiently. 02/02/2016, Bing Li
84          this.clientPool = new FreeClientPool(ChatConfig.CLIENT_POOL_SIZE);
85          // Set idle checking for the client pool. 04/17/2017, Bing Li
86          this.clientPool.setIdleChecker(ChatConfig.CLIENT_IDLE_CHECK_DELAY,
87              ChatConfig.CLIENT_IDLE_CHECK_PERIOD, ChatConfig.CLIENT_MAX_IDLE_TIME);
88
89          // Initialize the thread pool. 05/27/2017, Bing Li
90          this.pool = new ThreadPool(ChatConfig.EVENTER_THREAD_POOL_SIZE,
91              ChatConfig.EVENTER_THREAD_POOL_ALIVE_TIME);
92
93          // Initialize the asynchronous eventer for ShutdownChatServerNotification. 04/27/2017, Bing Li
94          this.shutdownChattingServerNotificationEventer = new AsyncRemoteEventer
95              .AsyncRemoteEventerBuilder<ShutdownChatServerNotification>()
96              .threadPool(this.pool)
97              .clientPool(this.clientPool)
98              .eventQueueSize(ChatConfig.ASYNC_EVENT_QUEUE_SIZE)
99              .eventerSize(ChatConfig.ASYNC_EVENTER_SIZE)
```

```
100             .eventingWaitTime(ChatConfig.ASYNC_EVENTING_WAIT_TIME)
101             .eventerWaitTime(ChatConfig.ASYNC_EVENTER_WAIT_TIME)
102             .waitRound(ChatConfig.ASYNC_EVENTER_WAIT_ROUND)
103             .idleCheckDelay(ChatConfig.ASYNC_EVENT_IDLE_CHECK_DELAY)
104             .idleCheckPeriod(ChatConfig.ASYNC_EVENT_IDLE_CHECK_PERIOD)
105             .schedulerPoolSize(ClientConfig.SCHEDULER_POOL_SIZE)
106             .schedulerKeepAliveTime(ClientConfig.SCHEDULER_KEEP_ALIVE_TIME)
107             .build();
108
109         // Initialize the asynchronous eventer for ShutdownChildrenAdminNotification. 04/27/2017, Bing Li
110         this.shutdownClusterChildrenNotificationEventer = new AsyncRemoteEventer
111             .AsyncRemoteEventerBuilder<ShutdownChildrenAdminNotification>()
112             .threadPool(this.pool)
113             .clientPool(this.clientPool)
114             .eventQueueSize(ChatConfig.ASYNC_EVENT_QUEUE_SIZE)
115             .eventerSize(ChatConfig.ASYNC_EVENTER_SIZE)
116             .eventingWaitTime(ChatConfig.ASYNC_EVENTING_WAIT_TIME)
117             .eventerWaitTime(ChatConfig.ASYNC_EVENTER_WAIT_TIME)
118             .waitRound(ChatConfig.ASYNC_EVENTER_WAIT_ROUND)
119             .idleCheckDelay(ChatConfig.ASYNC_EVENT_IDLE_CHECK_DELAY)
120             .idleCheckPeriod(ChatConfig.ASYNC_EVENT_IDLE_CHECK_PERIOD)
121             .schedulerPoolSize(ClientConfig.SCHEDULER_POOL_SIZE)
122             .schedulerKeepAliveTime(ClientConfig.SCHEDULER_KEEP_ALIVE_TIME)
123             .build();
124     }
125
126     /*
127      * Notify the chatting server to shut down. 04/17/2017, Bing Li
128      */
129     public void notifyShutdownCSChatServer()
130     {
131         // Check whether the eventer manager is ready to send asynchronous notifications. 04/17/2017, Bing Li
132         if (!this.shutdownChattingServerNotificationEventer.isReady())
133         {
134             // Execute the eventer manager. 04/17/2017, Bing Li
135             this.pool.execute(this.shutdownChattingServerNotificationEventer);
136         }
137         // Send the notification. 04/17/2017, Bing Li
138         this.shutdownChattingServerNotificationEventer.notify(ChatConfig.CHAT_SERVER_ADDRESS,
139             ChatConfig.CHAT_ADMIN_PORT, new ShutdownChatServerNotification());
140     }
141
142     /*
143      * Notify the chatting registry server to shutdown. 05/01/2017, Bing Li
144      */
145     public void notifyShutdownChattingRegistryServer()
146     {
147         // Check whether the eventer manager is ready to send asynchronous notifications. 04/17/2017, Bing Li
148         if (!this.shutdownChattingServerNotificationEventer.isReady())
149         {
150             // Execute the eventer manager. 04/17/2017, Bing Li
151             this.pool.execute(this.shutdownChattingServerNotificationEventer);
152         }
153         // Send the notification. 04/17/2017, Bing Li
154         this.shutdownChattingServerNotificationEventer.notify(ChatConfig.CHAT_REGISTRY_ADDRESS,
155             ChatConfig.CHAT_ADMIN_PORT, new ShutdownChatServerNotification());
156     }
157
158     /*
159      * Notify the cluster root to shutdown cluster children. 05/20/2017, Bing Li
160      */
161     public void notifyShutdownClusterChildren()
162     {
163         // Check whether the eventer manager is ready to send asynchronous notifications. 04/17/2017, Bing Li
164         if (!this.shutdownClusterChildrenNotificationEventer.isReady())
165         {
166             // Execute the eventer manager. 04/17/2017, Bing Li
167             this.pool.execute(this.shutdownClusterChildrenNotificationEventer);
168         }
169
170         // Get the cluster root's IP and port. 05/20/2017, Bing Li
171         ChatPartnerResponse response = ChatAdminReader.RR().searchUser
172             (ClusterConfig.CLUSTER_ROOT_KEY);
173
174         // Send the notification. 04/17/2017, Bing Li
175         this.shutdownClusterChildrenNotificationEventer.notify(response.getIP(), response.getPort(),
176             new ShutdownChildrenAdminNotification());
```

```
177         }
178     }
```

List 5.53 The code of ChatAdminEventer.java

Line 26~27 declares the instance, shutdownChattingServerNotificationEventer, of AsyncRemoteEventer.

Line 30~31 declares the instance, shutdownClusterChildrenNotificationEventer, of AsyncRemoteEventer.

Line 34 declares the instance of a thread pool, ThreadPool.

Line 37 declare the instance of a TCP client pool, FreeClientPool.

The RE is usually a singleton such that it is defined through Line 39~59.

Line 64~76 disposes all of claimed resources in the RE.

Line 81~124 initializes each of the declared resources between Line 26~37. Many parameters should be set when doing that. You can refer to Line 66~105 in List 5.43 of ChatEventer.java and Table 5.7.

Line 129~140 sends the notification of ShutdownChatServerNotification to the chatting server asynchronously. That is what we need in our current chatting system.

Line 145~156 sends the notification of ShutdownChatServerNotification to the registry server. Unfortunately, until now, you must have no idea what the registry server is. Just leave the lines here and we will elaborate to you later.

Line 161~177 sends another notification of ShutdownChildrenAdminNotification to the root of a cluster. You must get confused about that. Do not worry. This is our advanced topic to be taught.

4.3 RR: The Reader of the Administrator

In the chatting system implemented with C/S, the reader is not useful. However, since it first emerges in Line 29 of the administration client, the Administrator, I just list it here. We will discuss with you in the later chapter. Table 5.29 summarizes the pattern of RR for the administrator.

Pattern: RR (Remote Reader)		
Class	ChatAdminReader	
Singleton	Yes	
Method	void init()	
	void shutdown()	
	ChatPartnerResponse searchUser(String userKey)	
Enclosed Pattern	Invocation	RemoteReader.read(String nodeKey, String ip, int port, ServerMessage request)
Employed API	RemoteReader	

Table 5.29 The pattern of RR for the administrator

```
1    package com.greatfree.chat.admin;
2
3    import java.io.IOException;
4
5    import com.greatfree.chat.ChatConfig;
6    import com.greatfree.chat.message.ChatMessageConfig;
7    import com.greatfree.chat.message.peer.ChatPartnerRequest;
8    import com.greatfree.chat.message.peer.ChatPartnerResponse;
9    import com.greatfree.exceptions.RemoteReadException;
10   import com.greatfree.remote.RemoteReader;
11   import com.greatfree.testing.data.ClientConfig;
12   import com.greatfree.util.NodeID;
13   import com.greatfree.util.ServerStatus;
14
15   /*
16    * The class wraps the class, RemoteReader, to send IP requests to the register server to get IPs to
17    * the servers or peers to be managed. 04/23/2017, Bing Li
18    */
19
20   // Created: 05/20/2017, Bing Li
21   public class ChatAdminReader
22   {
23       private ChatAdminReader()
24       {
25       }
26
27       /*
28        * Initialize a singleton. 11/27/2014, Bing Li
29        */
30       private static ChatAdminReader instance = new ChatAdminReader();
31
32       public static ChatAdminReader RR()
33       {
34           if (instance == null)
35           {
36               instance = new ChatAdminReader();
37               return instance;
38           }
39           else
40           {
41               return instance;
42           }
43       }
44
45       /*
46        * Initialize the remote reader to send requests and receive responses from the remote
47        * server. 11/23/2014, Bing Li
48        */
49       public void init()
50       {
51           RemoteReader.REMOTE().init(ClientConfig.CLIENT_READER_POOL_SIZE);
52       }
53
54       /*
55        * Shutdown the remote reader. 11/23/2014, Bing Li
56        */
57       public void shutdown()
58       {
59           try
60           {
61               RemoteReader.REMOTE().shutdown();
62           }
63           catch (IOException e)
64           {
65               ServerStatus.FREE().printException(e);
66           }
67       }
68
69       /*
70        * Send the request the chatting registry server to retrieve the specified registered chatting
71        * partner. 05/28/2017, Bing Li
72        */
73       public ChatPartnerResponse searchUser(String userKey)
74       {
75           try
76           {
77               return (ChatPartnerResponse)(RemoteReader.REMOTE().read(NodeID.DISTRIBUTED().getKey(),
```

```
78                  ChatConfig.CHAT_REGISTRY_ADDRESS, ChatConfig.CHAT_REGISTRY_PORT, new
79                  ChatPartnerRequest(userKey)));
80          }
81          catch (ClassNotFoundException | RemoteReadException | IOException e)
82          {
83              e.printStackTrace();
84          }
85          return ChatMessageConfig.NO_PEER_CHAT_PARTNER_RESPONSE;
86      }
87  }
```

List 5.54 The code of ChatAdminReader.java

4.4 The UI

With the support of GreatFree's RR and RE, it is convenient to implement the client. Now the code is almost done. The last one to show you is the one to show the menu, i.e., a simple UI, which is shown in the code of ChatMenu.java in List 5.58.

```
1   package com.greatfree.chat.admin;
2
3   import com.greatfree.admin.Menu;
4
5   /*
6    * This is a menu for the administrator to display a control interface. 04/17/2017, Bing Li
7    */
8
9   // Created: 04/17/2017, Bing Li
10  public class ChatMenu
11  {
12      public final static String STOP_CHATTING_SERVER = Menu.TAB +
13          "1) Stop CS Chatting Server";
14      public final static String STOP_CHATTING_REGISTRY_SERVER = Menu.TAB +
15          "2) Stop Chatting Registry Server";
16      public final static String STOP_CLUSTER_CHILDREN = Menu.TAB +
17          "3) Stop Cluster Children";
18  }
```

List 5.55 The code of ChatMenu.java

5. The Configurations

Many parameters should be set when initializing APIs of GreatFree. You can see the most important ones in Table 5.3~5.7. In the simple chatting system, those parameters are defined in the code of ChatConfig.java, ClientConfig.java and ServerConfig.java, respectively. For the sample code, it is fine to define parameters in this way. But to deploy a practical system, you are highly suggested to set up those parameters out of the code, such as defining them in XML files [8]. Then, you can update those parameters without compilation.

5.1 Chatting Configurations

For the chatting system, most parameters are defined in the code ChatConfig.java, as shown in List 5.56.

```
1   package com.greatfree.chat;
2
```

```
3    import com.greatfree.util.Tools;
4
5    /*
6     * The class is the entry to the server process. 08/22/2014, Bing Li
7     */
8
9    // Created: 04/15/2017, Bing Li
10   public class ChatConfig
11   {
12       public final static String CHAT_SERVER_ADDRESS = "192.168.1.108";
13       public final static String CHAT_REGISTRY_ADDRESS = "192.168.1.104";
14       public final static String CHAT_SERVER_KEY = Tools.generateUniqueKey();
15       public final static String CHAT_REGISTRY_SERVER_KEY = Tools.generateUniqueKey();
16       public final static String CHAT_REGISTRY_NAME = Tools.generateUniqueKey();
17       public final static String PEER_REGISTRY_KEY = Tools.generateUniqueKey();
18       public final static String PEER_ADMIN_PORT_KEY = Tools.generateUniqueKey();
19       public final static String CHAT_REGISTRY_PORT_KEY = Tools.generateUniqueKey();
20       public final static String REGISTRY_ADMIN_PORT_KEY = Tools.generateUniqueKey();
21
22       public final static int CHAT_SERVER_PORT = 8944;
23       public final static int CHAT_ADMIN_PORT = 8943;
24       public final static int CHAT_REGISTRY_PORT = 8942;
25
26       public final static int DISPATCHER_THREAD_POOL_SIZE = 500;
27       public final static long DISPATCHER_THREAD_POOL_KEEP_ALIVE_TIME = 2000;
28
29       public final static int CLIENT_POOL_SIZE = 500;
30       public final static long CLIENT_IDLE_CHECK_DELAY = 3000;
31       public final static long CLIENT_IDLE_CHECK_PERIOD = 5000;
32       public final static long CLIENT_MAX_IDLE_TIME = 3000;
33
34       public final static int EVENTER_THREAD_POOL_SIZE = 10;
35       public final static long EVENTER_THREAD_POOL_ALIVE_TIME = 10000;
36
37       public final static int ASYNC_EVENT_QUEUE_SIZE = 100;
38       public final static int ASYNC_EVENTER_SIZE = 10;
39       public final static int ASYNC_EVENTING_WAIT_TIME = 1000;
40       public final static int ASYNC_EVENTER_WAIT_TIME = 1000;
41       public final static int ASYNC_EVENTER_WAIT_ROUND = 5;
42       public final static long ASYNC_EVENT_IDLE_CHECK_DELAY = 3000;
43       public final static long ASYNC_EVENT_IDLE_CHECK_PERIOD = 3000;
44
45       public final static int SCHEDULER_THREAD_POOL_SIZE = 50;
46       public final static long SCHEDULER_THREAD_POOL_KEEP_ALIVE_TIME = 5000;
47
48       public final static int STOP_CS_CHATTING_SERVER = 1;
49       public final static int STOP_CHATTING_REGISTRY_SERVER = 2;
50       public final static int STOP_CLUSTER_CHILDREN = 3;
51
52       public final static long CHAT_POLLING_DELAY = 0;
53       public final static long CHAT_POLLING_PERIOD = 10000;
54
55       public final static int READER_CLIENT_SIZE = 100;
56       public final static long SYNC_EVENTER_IDLE_CHECK_DELAY = 4000;
57       public final static long SYNC_EVENTER_IDLE_CHECK_PERIOD = 5000;
58       public final static long SYNC_EVENTER_MAX_IDLE_TIME = 3000;
59       public final static int CLIENT_THREAD_POOL_SIZE = 100;
60       public final static long CLIENT_THREAD_KEEP_ALIVE_TIME = 3000;
61   }
```

List 5.56 The code of ChatConfig.java

5.2 The Client Configurations

Some parameters are defined in the code of ClientConfig.java, as shown in List 5.57. It needs to indicate that not all of the parameters are utilized in the chatting system since the configuration is shared by other projects in our source code. The constants in ClientConfig is usually used to set up the parameters in the APIs that run on the client, such as the code of ChatEventer.java in List 5.38.

```
1    package com.greatfree.testing.data;
2
3    /*
4     * The class contains all of constants at the client end. 09/21/2014, Bing Li
5     */
6
7    // Created: 09/21/2014, Bing Li
8    public class ClientConfig
9    {
10       public final static int CLIENT_POOL_SIZE = 500;
11       public final static String USERNAME = "greatfree";
12       public final static String PASSWORD = "19890604";
13       public final static int CLIENT_LISTENER_THREAD_POOL_SIZE = 80;
14       public final static long CLIENT_LISTENER_THREAD_ALIVE_TIME = 10000;
15
16       public final static long CLIENT_IDLE_CHECK_DELAY = 3000;
17       public final static long CLIENT_IDLE_CHECK_PERIOD = 5000;
18       public final static long CLIENT_MAX_IDLE_TIME = 3000;
19
20       public final static int TIME_TO_WAIT_FOR_THREAD_TO_DIE = 10;
21
22       public final static int CLIENT_READER_POOL_SIZE = 10;
23
24       public final static int EVENTER_THREAD_POOL_SIZE = 10;
25       public final static long EVENTER_THREAD_POOL_ALIVE_TIME = 10000;
26
27       public final static int SERVER_IO_POOL_SIZE = 100;
28
29       public final static int MULTICASTOR_POOL_SIZE = 100;
30       public final static long MULTICASTOR_WAIT_TIME = 1000;
31       public static final int MULTICAST_BRANCH_COUNT = 16;
32
33       public final static int SCHEDULER_POOL_SIZE = 50;
34       public final static long SCHEDULER_KEEP_ALIVE_TIME = 5000;
35
36       public final static int ASYNC_EVENT_QUEUE_SIZE = 100;
37       public final static int ASYNC_EVENTER_SIZE = 10;
38       public final static int ASYNC_EVENTING_WAIT_TIME = 1000;
39       public final static int ASYNC_EVENTER_WAIT_TIME = 1000;
40       public final static int ASYNC_EVENTER_WAIT_ROUND = 5;
41       public final static long ASYNC_EVENT_IDLE_CHECK_DELAY = 3000;
42       public final static long ASYNC_EVENT_IDLE_CHECK_PERIOD = 3000;
43   }
```

List 5.57 The code of ClientConfig.java

5.3 The Server Configurations

Similarly, on the other hand, the code of ServerConfig.java, shown in List 5.58, defines most constants that are used to set up the APIs on the server side, such as those in the code of ChatServerDispatcher.java in List 5.8.

```
1    package com.greatfree.testing.data;
2
3    import com.greatfree.remote.FreeClient;
4
5    /*
6     * The class keeps constants of the testing sample code. 08/04/2014, Bing Li
7     */
8
9    // Created: 08/04/2014, Bing Li
10   public class ServerConfig
11   {
12       public final static String SERVER_IP = "192.168.1.104";
13       public final static int SERVER_PORT = 8964;
14       public final static int ADMIN_PORT = 8947;
15       public final static int CLIENT_PORT = 8949;
16
17       public final static int DISPATCHER_POOL_SIZE = 500;
18       public final static long DISPATCHER_POOL_THREAD_POOL_ALIVE_TIME = 2000;
```

```
19
20      public final static int LISTENING_THREAD_COUNT = 5;
21      public final static int SINGLE_THREAD_COUNT = 1;
22      public final static int LISTENER_THREAD_POOL_SIZE = 50;
23      public final static long LISTENER_THREAD_ALIVE_TIME = 10000;
24
25      public final static int MAX_SERVER_IO_COUNT = 500;
26
27      public final static long TERMINATE_SLEEP = 2000;
28
29      public final static FreeClient NO_CLIENT = null;
30
31      public final static int MY_SERVER = 1;
32
33      public final static int MAX_CLIENT_LISTEN_THREAD_COUNT = 5;
34
35      public final static String ROOT_PATH = "/home/libing/GreatFreeLabs/";
36      public final static String CONFIG_HOME = ServerConfig.ROOT_PATH + "Config/";
37
38      public final static String COORDINATOR_ADDRESS = "127.0.0.1";
39      public final static int COORDINATOR_PORT_FOR_CRAWLER = 8963;
40      public final static int COORDINATOR_PORT_FOR_MEMORY = 8965;
41      public final static int COORDINATOR_PORT_FOR_ADMIN = 8951;
42      public final static int COORDINATOR_PORT_FOR_SEARCH = 8950;
43      public final static int CRAWL_SERVER_PORT = 8960;
44      public final static int MEMORY_SERVER_PORT = 8961;
45      public final static int SEARCH_CLIENT_PORT = 8948;
46      public final static int COORDINATOR_DN_PORT = 8946;
47      public final static int DN_PORT = 8945;
48
49      public final static long REQUEST_THREAD_WAIT_TIME = 1000;
50      public final static long NOTIFICATION_THREAD_WAIT_TIME = 1000;
51
52      public final static int REQUEST_DISPATCHER_POOL_SIZE = 50;
53      public final static long REQUEST_DISPATCHER_THREAD_ALIVE_TIME = 500;
54      public final static int MAX_REQUEST_TASK_SIZE = 500;
55      public final static int MAX_REQUEST_THREAD_SIZE = 50;
56      public final static long REQUEST_DISPATCHER_WAIT_TIME = 1000;
57      public final static int REQUEST_DISPATCHER_WAIT_ROUND = 5;
58      public final static long REQUEST_DISPATCHER_IDLE_CHECK_DELAY = 2000;
59      public final static long REQUEST_DISPATCHER_IDLE_CHECK_PERIOD = 2000;
60
61      public final static int NOTIFICATION_DISPATCHER_POOL_SIZE = 30;
62      public final static long NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME = 2000;
63
64      public final static int MAX_NOTIFICATION_TASK_SIZE = 500;
65      public final static int MAX_NOTIFICATION_THREAD_SIZE = 10;
66
67      public final static long NOTIFICATION_DISPATCHER_WAIT_TIME = 1000;
68      public final static int NOTIFICATION_DISPATCHER_WAIT_ROUND = 5;
69      public final static long NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY = 3000;
70      public final static long NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD = 3000;
71
72      public final static int CLIENT_POOL_SIZE = 500;
73      public final static long CLIENT_IDLE_CHECK_DELAY = 3000;
74      public final static long CLIENT_IDLE_CHECK_PERIOD = 30000;
75      public final static long CLIENT_MAX_IDLE_TIME = 3000;
76
77      public final static long DISTRIBUTE_DATA_WAIT_TIME = 2000;
78      public final static int MULTICASTOR_POOL_SIZE = 100;
79      public final static long MULTICASTOR_POOL_WAIT_TIME = 1000;
80
81      public final static int ROOT_MULTICAST_BRANCH_COUNT = 100;
82      public final static int MULTICAST_BRANCH_COUNT = 16;
83
84      public final static int SCHEDULER_POOL_SIZE = 50;
85      public final static long SCHEDULER_KEEP_ALIVE_TIME = 5000;
86
87      public final static int SHARED_THREAD_POOL_SIZE = 50;
88      public final static long SHARED_THREAD_POOL_KEEP_ALIVE_TIME = 5000;
89  }
```

List 5.58 The code of ServerConfig.java

6. Testing

We have talked about the development environment in Chapter 2. So, I will not present the steps in detail when testing the chatting system. If you are not familiar with the deployment procedure, you can refer to Chapter 2.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatRegistry — ssh — 85×40
x ./util/XMLConstants.java
x ./util/XMLReader.java
x ./util/XPathOnDiskReader.java
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatServer/src/com/greatfree$ cd ..
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatServer/src/com$ cd ..
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatServer/src$ cd ..
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatServer$ ls
build           build.xml       lib             src
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatServer$ ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
        options=3<RXCSUM,TXCSUM>
        inet6 ::1 prefixlen 128
        inet 127.0.0.1 netmask 0xff000000
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
        nd6 options=1<PERFORMNUD>
gif0: flags=8010<POINTOPOINT,MULTICAST> mtu 1280
stf0: flags=0<> mtu 1280
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        options=27<RXCSUM,TXCSUM,VLAN_MTU,TSO4>
        ether 00:26:b0:ed:f2:10
        nd6 options=1<PERFORMNUD>
        media: autoselect
        status: inactive
en1: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        ether f8:1e:df:df:91:0d
        inet6 fe80::fa1e:dfff:fedf:910d%en1 prefixlen 64 scopeid 0x5
        inet 192.168.1.105 netmask 0xffffff00 broadcast 192.168.1.255
        nd6 options=1<PERFORMNUD>
        media: autoselect
        status: active
fw0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 4078
        lladdr 00:26:b0:ff:fe:ed:f2:10
        nd6 options=1<PERFORMNUD>
        media: autoselect <full-duplex>
        status: inactive
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
        ether 0a:1e:df:df:91:0d
        media: autoselect
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatServer$
```

Figure 5.1 The chatting server terminal is ready for testing

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
libing@freeweb:~/GreatFreeLabs/Clouds/ChatClient$ ls
build  build.xml  lib  src
libing@freeweb:~/GreatFreeLabs/Clouds/ChatClient$ cd src/com/greatfree/
libing@freeweb:~/GreatFreeLabs/Clouds/ChatClient/src/com/greatfree$ ls
admin  chat     concurrency  message    remote  server   tutorial
cache  cluster  exceptions   multicast  reuse   testing  util
libing@freeweb:~/GreatFreeLabs/Clouds/ChatClient/src/com/greatfree$ cd ..
libing@freeweb:~/GreatFreeLabs/Clouds/ChatClient/src/com$ ls
greatfree
libing@freeweb:~/GreatFreeLabs/Clouds/ChatClient/src/com$ cd ..
libing@freeweb:~/GreatFreeLabs/Clouds/ChatClient/src$ cd ..
libing@freeweb:~/GreatFreeLabs/Clouds/ChatClient$ ls
build  build.xml  lib  src
libing@freeweb:~/GreatFreeLabs/Clouds/ChatClient$ ifconfig
eth0      Link encap:Ethernet  HWaddr b8:ac:6f:92:ef:ef
          inet addr:192.168.1.112  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::baac:6fff:fe92:efef/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:137932 errors:0 dropped:0 overruns:0 frame:0
          TX packets:83768 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:164195592 (164.1 MB)  TX bytes:9324844 (9.3 MB)

eth1      Link encap:Ethernet  HWaddr b8:ac:6f:92:ef:f0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:126 errors:0 dropped:0 overruns:0 frame:0
          TX packets:126 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:10328 (10.3 KB)  TX bytes:10328 (10.3 KB)

libing@freeweb:~/GreatFreeLabs/Clouds/ChatClient$
```

Figure 5.2 One chatting client terminal is ready for testing

6.1 Starting Up

To test the system in a more practical environment, four PCs are employed. They play the roles, the chatting server, two chatting clients and the chatting administrator, respectively. After deploying the code, you should have four terminals for each of the PCs, as shown in Figure 5.1, Figure 5.2, Figure 5.3 and Figure 5.4.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
./util/SerializedKey.java
./util/ServerStatus.java
./util/StringObj.java
./util/Symbols.java
./util/TerminateSignal.java
./util/Time.java
./util/Timing.java
./util/Tools.java
./util/UtilConfig.java
./util/XMLConstants.java
./util/XMLReader.java
./util/XPathOnDiskReader.java
libing@mother:~/GreatFreeLabs/Clouds/ChatClient/src/com/greatfree$ ls
admin  concurrency  message   remote  server   tutorial
cache  exceptions   multicast reuse   testing  util
libing@mother:~/GreatFreeLabs/Clouds/ChatClient/src/com/greatfree$ cd ..
libing@mother:~/GreatFreeLabs/Clouds/ChatClient/src/com$ cd ..
libing@mother:~/GreatFreeLabs/Clouds/ChatClient/src$ cd ..
libing@mother:~/GreatFreeLabs/Clouds/ChatClient$ ls
build  build.xml  lib  src
libing@mother:~/GreatFreeLabs/Clouds/ChatClient$ ifconfig
eth0      Link encap:Ethernet  HWaddr 9c:5c:8e:bf:50:6b
          inet addr:192.168.1.108  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::2f64:b495:9e68:c084/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:113188 errors:0 dropped:0 overruns:0 frame:0
          TX packets:60187 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:156776637 (156.7 MB)  TX bytes:6966998 (6.9 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:221 errors:0 dropped:0 overruns:0 frame:0
          TX packets:221 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:16554 (16.5 KB)  TX bytes:16554 (16.5 KB)

libing@mother:~/GreatFreeLabs/Clouds/ChatClient$
```

Figure 5.3 Another chatting client terminal is ready for testing

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatAdmin — ssh — 85×40
admin  cluster      exceptions  remote  testing
cache  clustering   message     reuse   tutorial
chat   concurrency  multicast   server  util
libing@freescape:~/GreatFreeLabs/Clouds/ChatAdmin/src/com/greatfree$ cd ..
libing@freescape:~/GreatFreeLabs/Clouds/ChatAdmin/src/com$ cd ..
libing@freescape:~/GreatFreeLabs/Clouds/ChatAdmin/src$ ls
com
libing@freescape:~/GreatFreeLabs/Clouds/ChatAdmin/src$ cd ..
libing@freescape:~/GreatFreeLabs/Clouds/ChatAdmin$ ls
build  build.xml  lib  src
libing@freescape:~/GreatFreeLabs/Clouds/ChatAdmin$ ifconfig
eth0      Link encap:Ethernet  HWaddr f0:de:f1:45:df:be
          inet addr:192.168.1.109  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::f2de:f1ff:fe45:dfbe/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:192014 errors:0 dropped:0 overruns:0 frame:0
          TX packets:336582 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:20001499 (20.0 MB)  TX bytes:474911303 (474.9 MB)
          Interrupt:20 Memory:f2600000-f2620000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:171 errors:0 dropped:0 overruns:0 frame:0
          TX packets:171 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:14055 (14.0 KB)  TX bytes:14055 (14.0 KB)

wlan0     Link encap:Ethernet  HWaddr 58:94:6b:a5:05:a4
          inet addr:192.168.1.104  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::5a94:6bff:fea5:5a4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:787 errors:0 dropped:0 overruns:0 frame:0
          TX packets:64 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:97543 (97.5 KB)  TX bytes:12327 (12.3 KB)

libing@freescape:~/GreatFreeLabs/Clouds/ChatAdmin$
```

Figure 5.4 The chatting administrator terminal is ready for testing

According to the terminals, you can see that each of them has their own IPs. If you do not has sufficient number of PCs, it does not matter. For example, if you have only one PC, you just need to change the IP address of the chatting server, CHAT_SERVER_ADDRESS, to the local IP address, 127.0.0.1, i.e., Line 12 of ChatConfig.java in List 5.56. You should pay much attention to the IP address of the chatting server.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatRegistry — ssh — 85×40
        media: autoselect
        status: inactive
en1: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        ether f8:1e:df:df:91:0d
        inet6 fe80::fa1e:dfff:fedf:910d%en1 prefixlen 64 scopeid 0x5
        inet 192.168.1.105 netmask 0xffffff00 broadcast 192.168.1.255
        nd6 options=1<PERFORMNUD>
        media: autoselect
        status: active
fw0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 4078
        lladdr 00:26:b0:ff:fe:ed:f2:10
        nd6 options=1<PERFORMNUD>
        media: autoselect <full-duplex>
        status: inactive
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
        ether 0a:1e:df:df:91:0d
        media: autoselect
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatServer$ ant
Buildfile: /GreatFreeLabs/Clouds/ChatServer/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ChatServer/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatServer/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatServer/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatServer/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ChatServer/build/clas
ses

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ChatServer/build/jar/FreeWeb.jar

run:
     [java] Chatting server starting up ...
     [java] Chatting server started ...
```

Figure 5.5 The chatting server is started up

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
          RX packets:126 errors:0 dropped:0 overruns:0 frame:0
          TX packets:126 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:10328 (10.3 KB)  TX bytes:10328 (10.3 KB)

libing@freeweb:~/GreatFreeLabs/Clouds/ChatClient$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ChatClient/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ChatClient/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatClient/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatClient/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatClient/build/jar

compile:
    [javac] Compiling 781 source files to /home/libing/GreatFreeLabs/Clouds/ChatClien
t/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatClient/build/jar/Free
Web.jar

run:
     [java] Tell me your user name:
gates
     [java] Tell me your partner:
greatfree
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 5.6 One chatting client, gates, is started

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40

main:

BUILD SUCCESSFUL
Total time: 18 seconds
libing@mother:~/GreatFreeLabs/Clouds/ChatClient$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ChatClient/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ChatClient/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatClient/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatClient/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatClient/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ChatClien
t/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatClient/build/jar/Free
Web.jar

run:
     [java] Tell me your user name:
greatfree
     [java] Tell me your partner:
gates
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 5.7 Another chatting client, greatfree, is started up

Now let us take a look how the chatting system works. First of all, the chatting server needs to be started up before the chatting clients. That is easy to understand since the chatting is performed with the support of the server. Figure 5.5 shows the chatting server is started.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatRegistry — ssh — 85×40
clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ChatServer/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatServer/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatServer/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatServer/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ChatServer/build/clas
ses

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ChatServer/build/jar/FreeWeb.jar

run:
     [java] Chatting server starting up ...
     [java] Chatting server started ...
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:34:31 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:34:38 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:34:44 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:34:48 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:34:52 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:34:58 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:35:02 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:35:08 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:35:12 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:35:18 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:35:22 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:35:28 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:35:32 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:35:38 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:35:42 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:35:48 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:35:53 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:35:58 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:36:02 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:36:08 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:36:12 PDT 2017
```

Figure 5.8 The chatting server receives the polling request for new sessions

After the chatting server is started, it is time to start the chatting clients. In our current system, two clients participate in the chatting. When starting them, you do not need to consider the order. When a chatting client is started, you are asked to specify a username and your potential partner for chatting, as shown in Figure 5.6. In this case, the local username is gates whereas the potential partner is greatfree. To be convenient to explain the testing, the two chatting clients are named as gates and greatfree, respectively.

In contrast, you can start another chatting client. After it is started, you need to specific the username and the potential partner as well. In this case, the username is greatfree and the potential chatting partner is gates. Figure 5.7 illustrates the procedure.

After the two chatting clients are started, the terminal of the chatting server is updated periodically. It shows that the server receives the request of POLL_NEW_SESSIONS_REQUEST, as shown in Figure 5.8.

Before performing chatting, it is time to start the chatting administrator. Figure 5.9 demonstrates the scenario, in which a simple menu is displayed. You do not need to take care of the other three at this time. Only the first one makes sense to use in the chatting system.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatAdmin — ssh — 85×40

wlan0     Link encap:Ethernet  HWaddr 58:94:6b:a5:05:a4
          inet addr:192.168.1.104  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::5a94:6bff:fea5:5a4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4531 errors:0 dropped:0 overruns:0 frame:0
          TX packets:68 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:442114 (442.1 KB)  TX bytes:12763 (12.7 KB)

libing@freescape:~/GreatFreeLabs/Clouds/ChatAdmin$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ChatAdmin
/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/jar/FreeW
eb.jar

run:
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Stop CS Chatting Server
     [java]     2) Stop Chatting Registry Server
     [java]     3) Stop Cluster Children
     [java]     0) End
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 5.9 The chatting administrator is started up

6.2 Registering

Now it is time for us to start chatting. What you need to do is performed at the chatting clients. At the chatting client of gates, select the first option in the menu, for

example, “1) Register Chatting: gates”, as shown in Figure 5.10. At the same moment, the chatting server receives one new request, CS_CHAT_REGISTRY_REQUEST, as shown in Figure 5.11.

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatClient/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatClient/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ChatClien
t/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatClient/build/jar/Free
Web.jar

run:
     [java] Tell me your user name:
gates
     [java] Tell me your partner:
greatfree
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 1
     [java] Chatting registry status: true
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 5.10 At one chatting client, select the first option to register the chatting user, gates

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatRegistry — ssh — 85×40
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:50:28 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:50:32 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:50:38 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:50:42 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:50:48 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:50:52 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:50:58 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:51:02 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:51:08 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:51:12 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:51:19 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:51:22 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:51:28 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:51:32 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:51:38 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:51:43 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:51:48 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:51:52 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:51:58 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:52:02 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:52:08 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:52:12 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:52:18 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:52:22 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:52:28 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:52:32 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:52:38 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:52:42 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:52:48 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:52:52 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:52:58 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:53:02 PDT 2017
     [java] CS_CHAT_REGISTRY_REQUEST received @Sun May 28 14:53:02 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:53:08 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:53:12 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:53:18 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:53:22 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:53:28 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 14:53:32 PDT 2017
```

Figure 5.11 The chatting server receives one request, CS_CHAT_REGISTRY_REQUEST

Next, you need to perform the same registry for another chatting client, greatfree. After that, the greatfree client should be exactly like the one shown in Figure 5.12.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatClient/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatClient/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ChatClien
t/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatClient/build/jar/Free
Web.jar

run:
     [java] Tell me your user name:
greatfree
     [java] Tell me your partner:
gates
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 1
     [java] Chatting registry status: true
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 5.12 The chatting client of greatfree is registered after the first option, “Register Chatting: greatfree”, is selected

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
gates
     [java] Tell me your partner:
greatfree
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 1
     [java] Chatting registry status: true
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice: 2
     [java] greatfree: greatfree is a great & free guy!
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 5.13 The chatting client searches the potential partner, greatfree

After registering the chatting users, you can search your potential partners by selecting the second option in the gates client, i.e., “2) Search User: greatfree”, and

the search result is a simple description, “greatfree: greatfree is a great & free guy!”, as shown in Figure 5.13. And, on the greatfree terminal, you can perform the counterpart search, “2) Search User: gates”. If so, you can see the chatting UIs’ updates as shown in Figure 5.14 as well. Meanwhile, the chatting server should receive new requests, CS_CHAT_PARTNER_REQUEST, as shown in Figure 5.15.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
greatfree
     [java] Tell me your partner:
gates
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 1
     [java] Chatting registry status: true
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice: 2
     [java] gates: gates is a great & free guy!
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 5.14 Another chatting client searches its potential chatting partner

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatRegistry — ssh — 85×40
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:05:12 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:05:18 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:05:22 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:05:28 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:05:32 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:05:38 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:05:42 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:05:48 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:05:52 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:05:58 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:06:02 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:06:08 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:06:12 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:06:18 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:06:22 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:06:28 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:06:32 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:06:38 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:06:42 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:06:48 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:06:52 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:06:58 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:07:02 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:07:08 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:07:12 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:07:18 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:07:22 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:07:28 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:07:32 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:07:38 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:07:42 PDT 2017
     [java] CS_CHAT_PARTNER_REQUEST received @Sun May 28 15:07:46 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:07:48 PDT 2017
     [java] CS_CHAT_PARTNER_REQUEST received @Sun May 28 15:07:50 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:07:52 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:07:58 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:08:02 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:08:08 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:08:12 PDT 2017
```

Figure 5.15 The chatting server receives the request, CS_CHAT_PARTNER_REQUEST, to search chatting partners

6.3 Chatting

After registering chatting users, chatting can be performed. We are trying to imitate the practical product, such as Skype [9] or WeChat [10]. To chat, you need to add one friend. For that, you need to select the third option at the gates client side, i.e., "3) Add Friend: greatfree", as shown in Figure 5.16. Then, the chatting server gets updated for receiving the notification, CS_ADD_PARTNER_NOTIFICATION, as shown in Figure 5.17.

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
     [java] Input an option:
1
     [java] Your choice: 1
     [java] Chatting registry status: true
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice: 2
     [java] greatfree: greatfree is a great & free guy!
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
3
     [java] Your choice: 3
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 5.16 The chatting client adds its potential chatting partner, greatfree

You should notice that the UI of the chatting client of greatfree is also updated for its partner's operation of adding friends, as shown in Figure 5.18, in which it prompts that the chatting client detects one new chatting session is existed on the chatting server. The session is created because of the operation of adding friends. Since the existence of the chatting session is the precondition to perform chatting, you are ready for chatting at this moment.

If you take a look at the chatting server, besides the polling requests for new chatting sessions, new polling requests for new chatting messages, POLL_NEW_CHATS_REQUEST, are received by the server. The requests are sent by the greatfree client.

Now you need to add the partner of gates at the greatfree side, as shown in Figure 5.19. After that, the gates client detects the newly created chatting session, as shown in Figure 5.20.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatRegistry — ssh — 85×40
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:20:18 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:20:22 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:20:28 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:20:32 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:20:38 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:20:42 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:20:48 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:20:52 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:20:58 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:21:02 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:21:08 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:21:12 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:21:18 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:21:22 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:21:28 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:21:32 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:21:38 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:21:42 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:21:48 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:21:52 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:21:58 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:22:02 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:22:08 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:22:12 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:22:18 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:22:22 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:22:28 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:22:32 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:22:38 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:22:44 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:22:48 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:22:52 PDT 2017
     [java] CS_ADD_PARTNER_NOTIFICATION received @Sun May 28 15:22:56 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:22:58 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:23:02 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 15:23:02 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:23:09 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 15:23:12 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 15:23:12 PDT 2017
```

Figure 5.17 The chatting server receives the notification, CS_ADD_PARTNER_NOTIFICATION, and new polling requests, POLL_NEW_CHATS_REQUEST, after one chatting client adds one chatting friend

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
gates
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 1
     [java] Chatting registry status: true
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice: 2
     [java] gates: gates is a great & free guy!
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] 1 new chatting sessions are obtained ...
     [java] 1 new chatting sessions are retained ...
```

Figure 5.18 The chatting client of greatfree detects a new chatting session

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
     [java] Your choice: 1
     [java] Chatting registry status: true
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice: 2
     [java] :
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] 1 new chatting sessions are obtained ...
     [java] 1 new chatting sessions are retained ...
3
     [java] Your choice: 3
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 5.19 The greatfree client adds its partner gates as well

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
     [java] Tell me your partner:
greatfree
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice: 2
     [java] :
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
3
     [java] Your choice: 3
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] 1 new chatting sessions are obtained ...
     [java] 1 new chatting sessions are retained ...
```

Figure 5.20 The gates client detects a new chatting session

Until now, the chatting clients are ready for chatting. To do that, select the fourth option at the gates client, i.e., “4) Start Chatting with: greatfree”. After that, a sub menu is shown in the UI, as shown in Figure 5.21.

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice: 2
     [java] :
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
3
     [java] Your choice: 3
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] 1 new chatting sessions are obtained ...
     [java] 1 new chatting sessions are retained ...
4
     [java] Your choice: 4
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 5.21 The gates client starts chatting by selecting the fourth option and then a sub menu is displayed

You have to keep on selecting the first option of the sub menu. Then, the UI is updated to the one shown in Figure 5.22. To chat, you just simple type a message, for example, “hello”, as shown in Figure 5.23. Immediately after the gates client types the chatting message, “hello”, the chatting server should receive one new notification, CHAT_NOTIFICATION, as shown in 5.24.

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
2
     [java] Your choice: 2
     [java] :
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
3
     [java] Your choice: 3
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] 1 new chatting sessions are obtained ...
     [java] 1 new chatting sessions are retained ...
4
     [java] Your choice: 4
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 4
     [java] Please type your message:
```

Figure 5.22 The first option of the sub menu is selected

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
3
     [java] Your choice: 3
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
4     [java] 1 new chatting sessions are obtained ...
     [java] 1 new chatting sessions are retained ...

     [java] Your choice: 4
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 4
     [java] Please type your message:
hello
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 5.23 Type a chatting message, “hello”, at the gates client

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatRegistry — ssh — 85×40
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:52:17 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:52:20 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:52:27 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:52:30 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:52:37 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:52:40 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:52:47 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:52:50 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:52:57 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:53:00 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:53:07 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:53:10 PDT 2017
     [java] CS_CHAT_REGISTRY_REQUEST received @Sun May 28 16:53:11 PDT 2017
     [java] CS_CHAT_PARTNER_REQUEST received @Sun May 28 16:53:14 PDT 2017
     [java] CS_CHAT_PARTNER_REQUEST received @Sun May 28 16:53:17 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:53:17 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:53:20 PDT 2017
     [java] CS_ADD_PARTNER_NOTIFICATION received @Sun May 28 16:53:22 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:53:27 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 16:53:27 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:53:30 PDT 2017
     [java] CS_ADD_PARTNER_NOTIFICATION received @Sun May 28 16:53:33 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:53:37 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 16:53:37 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:53:40 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 16:53:40 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:53:47 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 16:53:47 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:53:50 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 16:53:50 PDT 2017
     [java] CHAT_NOTIFICATION received @Sun May 28 16:53:54 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:53:57 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 16:53:57 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:54:00 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 16:54:00 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:54:07 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 16:54:07 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 16:54:10 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 16:54:10 PDT 2017
```

Figure 5.24 The notification, CHAT_NOTIFICATION, is received at the chatting server immediately after the gates client enters the chatting message, “hello”

You must be anxious to know whether the “hello” message from the gates client can be shown in the greatfree client. The bad news is that the chatting message is not

shown immediately. However, the good news is that it is shown finally after a certain period, as shown in Figure 5.25. Since the greatfree client polls the chatting server every ten seconds. It might wait for ten seconds in maximum to get the "hello" message. If you are lucky enough, you do not even spend any time waiting. However, the average waiting time is five seconds.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice: 2
     [java] gates: gates is a great & free guy!
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
3
     [java] Your choice: 3
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] 1 new chatting sessions are obtained ...
     [java] 1 new chatting sessions are retained ...
     [java] PollNewChatsResponse: 1 messages received
     [java] gates says: hello
```

Figure 5.25 The greatfree client finally receives the "hello" message from the gates client

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] 1 new chatting sessions are obtained ...
     [java] 1 new chatting sessions are retained ...
3
     [java] Your choice: 3
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PollNewChatsResponse: 1 messages received
     [java] gates says: hello
4
     [java] Your choice: 4
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 4
     [java] Please type your message:
yes?
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 5.26 The greatfree client replies to the gates client by typing "yes?"

You can keep chatting. But this time, you play the role of greatfree. For example, you type “yes?” on the greatfree client, as shown in Figure 5.26. Similarly, the message, “yes?”, is received by the gates client after a certain period, as shown in Figure 5.27. Since the period of the gates is ten seconds, the delay of receiving chatting messages is identical to what happens at the greatfree client.

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
3
     [java] Your choice: 3
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] 1 new chatting sessions are obtained ...
     [java] 1 new chatting sessions are retained ...
4
     [java] Your choice: 4
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 4
     [java] Please type your message:
hello
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PollNewChatsResponse: 1 messages received
     [java] greatfree says: yes?
```

Figure 5.27 The gates client receives the chatting message, “yes?”, after a certain time

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 4
     [java] Please type your message:
hello
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PollNewChatsResponse: 1 messages received
     [java] greatfree says: yes?
0
     [java] Your choice: 4
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
0
     [java] Your choice: 0

main:

BUILD SUCCESSFUL
Total time: 14 minutes 10 seconds
libing@freeweb:~/GreatFreeLabs/Clouds/ChatClient$
```

Figure 5.28 Stop the chatting client of gates

Until now, it proves that the chatting system works! You can keep testing the system by typing more chatting messages.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatClient — ssh — 85×40
4
     [java] Your choice: 4
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 4
     [java] Please type your message:
yes?
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
0
     [java] Your choice: 4
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
0
     [java] Your choice: 0

main:

BUILD SUCCESSFUL
Total time: 14 minutes 12 seconds
libing@mother:~/GreatFreeLabs/Clouds/ChatClient$
```

Figure 5.29 Stop the chatting client of greatfree

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatAdmin — ssh — 85×40

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ChatAdmin
/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/jar/FreeW
eb.jar

run:
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Stop CS Chatting Server
     [java]     2) Stop Chatting Registry Server
     [java]     3) Stop Cluster Children
     [java]     0) End
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice is: 1
     [java]     1) Stop CS Chatting Server
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Stop CS Chatting Server
     [java]     2) Stop Chatting Registry Server
     [java]     3) Stop Cluster Children
     [java]     0) End
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 5.30 Select the option of “1) Stop CS Chatting Server” from the chatting administrator

6.4 Terminating

Now let us stop the system. As we discuss previously, a system should have the mechanism to stop itself well.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatRegistry — ssh — 85×40
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:18:36 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:18:39 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:18:39 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:18:46 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:18:46 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:18:49 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:18:49 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:18:56 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:18:56 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:18:59 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:18:59 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:19:06 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:19:06 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:19:09 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:19:09 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:19:16 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:19:16 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:19:19 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:19:19 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:19:26 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:19:26 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:19:29 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:19:29 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:19:36 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:19:36 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:19:39 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:19:39 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:19:46 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:19:46 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:19:49 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:19:49 PDT 2017
     [java] POLL_NEW_SESSIONS_REQUEST received @Sun May 28 17:19:56 PDT 2017
     [java] POLL_NEW_CHATS_REQUEST received @Sun May 28 17:19:56 PDT 2017
     [java] SHUTDOWN_CHAT_SERVER_NOTIFICATION received @Sun May 28 17:35:53 PDT 2017

main:

BUILD SUCCESSFUL
Total time: 30 minutes 8 seconds
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatServer$
```

Figure 5.31 The chatting server is stopped

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatAdmin — ssh — 85×40
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ChatAdmin
/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/jar/FreeW
eb.jar

run:
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Stop CS Chatting Server
     [java]     2) Stop Chatting Registry Server
     [java]     3) Stop Cluster Children
     [java]     0) End
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice is: 1
     [java]     1) Stop CS Chatting Server
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Stop CS Chatting Server
     [java]     2) Stop Chatting Registry Server
     [java]     3) Stop Cluster Children
     [java]     0) End
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
0
     [java] Your choice is: 0

main:

BUILD SUCCESSFUL
Total time: 31 minutes 47 seconds
libing@freescape:~/GreatFreeLabs/Clouds/ChatAdmin$
```

Figure 5.32 The chatting administrator is stopped

To do that, you can simply terminate the chatting clients by selecting its relevant menus since each of them has an option of "Quit" for you to stop. Figure 5.28 and Figure 5.29 show the procedure to stop the two chatting clients, respectively.

Since the chatting server has no menu option to stop itself, you have to do that with the administration server, which we start at the beginning of the testing. In Figure 5.9, there is a menu on the chatting administrator. You just simply select the first option, as shown in Figure 5.30.

The last step for the testing is to stop the administrator of the chatting. Since the administrator has a menu option for quitting, you just select it, as shown in Figure 5.32. After that, everything is done.

7 A Summary

Before moving forward to the next ones, let us make a brief summary about the code you have seen in the chapter. First of all, you need to program all of the code presented in the chapter yourself when you implement your applications with GreatFree. Those code can be your perfect samples for you to follow. In addition, you are required to learn fundamental knowledge about the distributed computing environment you will program. Otherwise, you cannot make an appropriate decision for it when encountering specific distributed problems. Likewise, because of APIs and idioms in GreatFree, the coding procedure is straightforward and convenient even though you need to retain technical knowledge. Finally, the chatting system is a typical and popular Client/Server distributed model. It can be used in broad areas of applications. If you work on the same distributed environment, you can even customize your systems with the coding in the chapter without coding from scratch. If so, your programming task is exactly the same to those discussed in Chapter 3 and Chapter 4.

References

[1] George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair. 2011. Processes, Chapter 3, Distributed Systems: Concepts and Design, the Fifth Edition.

[2] Elliotte Rusty Harold. 2014. Java Network Programming. O'Reilly Media, ISBN: 978-1-449-35767-2.

[3] Steven John Metsker. Design Patterns Java Workbook. Addison Wesley, 2002, ISBN: 0-201-74397-3

[4] Tomcat. 2015. Apache Tomcat. DOI: http://tomcat.apache.org/.

[5] JBoss. 2015. DOI: http://www.jboss.org/.

[6] Apache Web Server. DOI: https://httpd.apache.org/.

[7] Joshua Block. Effective Java, Section Edition. Addison-Wesley, 2008, ISBN-13: 978-0-321-35668-0, ISBN-10: 0-321-35668-3.

[8] Brett McLaughlin. Java and XML Data Binding. O'Reilly, 2002, ISBN: 0-596-00278-5.

[9] Skype. DOI: http://www.skype.com.

[10] WeChat. DOI: http://web.wechat.com.

Chapter 6 Programming with Peer

Abstract

Peer is another important distributed component that forms the distributed model of Peer-to-Peer (P2P) rather than that of Client/Server (C/S). When the system's scale is small and the distributed environment is located in a public computing context, developers should consider using the C/S architecture to implement their applications. However, the C/S intends to become a centralized system based on the server side. It is actually a barrier to program a scalable distributed system. For that, when the scale of the system should be large and the distributed environment is trustable or private, it is highly suggested to program with another distributed model, i.e., the peer.

The component of a peer can be roughly regarded as the combination of the client and the server. That is, a peer is able not only to send notifications and requests to a server but also to receive notifications and requests from any other clients. If such a peer is employed in a distributed system, distributed nodes can interact with each other arbitrarily without any limits. Then, such a system intends to be a decentralized one instead of a centralized one because interactions are performed without the server as a broker. So the scale of the entire system can be enlarged even to an infinite one. Moreover, when forming a large-scale distributed system, it is necessary to organize all of the nodes in a certain topology. If the node is a peer, it can be placed in any places. However, if a node is either a client or a server, it is difficult to construct the topology since no any interactions can be performed between a pair nodes when both of them are same type of nodes.

Although there are many advantages to program in a P2P model, it also has some disadvantages as well. Such a model cannot be widely used in a non-trustable environment since an arbitrary remote notification or request might be malicious. Additionally, as a decentralized system, the overhead to maintain such a system should be much higher than that of a centralized one. On the other hand, if computing nodes within a system are heterogeneous in terms of the computing resources they provide, the final system still has to be established as a centralized one, in which the powerful one is always the center.

However, a typical computing environment is easily to be implemented when using peer. That is the cluster, which is the technique that aggregates a large number of computer together such that they can work together to provide rich computing resources. Usually, a cluster is composed inside an organization. So it is not necessary to worry about the security issue. Likewise, within an organization, the computing environment is good and each computer holds similar amount of computing capacity. Therefore, a decentralized system can be formed conveniently. Finally, since the interactions among peers are free, it is convenient for a coordinator to place any possible topologies according the requirements in the real world. In short, if programming a cluster, the component of peer is a feasible and essential choice.

For that, GreatFree provides a new distributed component, Peer, other than the Client and the Server. Furthermore, a couple of code-level patterns, i.e., idioms, are also created by GreatFree such that your programming procedures become comfortable.

1. The Peer-Based Chatting System

A peer is a distributed component that not only plays the role of a client that sends notifications and requests actively but also receives notifications and requests passively as a server. Compared with the distributed components of clients and servers, it is believed to be the aggregation of both of them. In general, a peer is more convenient for developers to construct a distributed system. However, in practice, many systems are established with the components of clients and servers. That is not the result of techniques but due to the reasons of specific requirements and constraints of computing environments. On the other hand, when the environment appropriate to the peer exists, your systems turn out to be more scalable than those implemented by clients and servers.

For its importance, a new API, Peer, is proposed by GreatFree. Since we need to teach you how to program, the best way to achieve the goal is to give you some code that you can exercise. Similar to Chapter 5, the chatting system is programmed again in this chapter. But for this time, the fundamental component is updated from clients and servers to peers. Moreover, the distributed model is also changed from the C/S one to the P2P one. However, as we discuss in Chapter 5, because of the characters of the computing environment, most chatting systems are implemented by the C/S model. If so, why do we need to program it again? Just show you how to use the API of Peer? No, we have additional goals to teach you the topic of programming chatting with Peer. First of all, you can learn the API of Peer. It is critical without doubt. More important, the chatting system over P2P represents a generic distributed architecture that exists in all of the distributed systems, including the ones based on the C/S model. If the Peer-based chatting is available, you must learn the techniques of distributed systems deeply. The last benefit is that you can implement your own applications over the P2P chatting system rapidly by CPR.

Constructors	Explanation
Peer(Peer.PeerBuilder<Dispatcher> builder)	The constructor is implemented by the builder pattern
Peer(String peerName, String sysRegistryIP, int sysRegistryPort, int listenerCount, int listenerThreadPoolSize, long listenerThreadKeepAliveTime, Dispatcher dispatcher, int clientPoolSize, int readerClientSize, long syncEventerIdleCheckDelay, long syncEventerIdleCheckPeriod, long syncEventerMaxIdleTime, int asyncEventQueueSize, int asyncEventerSize, long asyncEventingWaitTime, long asyncEventerWaitTime, int asyncEventerWaitRound, long asyncEventIdleCheckDelay, long asyncEventIdleCheckPeriod, int clientThreadPoolSize, long clientThreadKeepAliveTime, int schedulerPoolSize, long schedulerKeepAliveTime)	This is the regular constructor to initialize an instance of Peer

Table 6.1 The constructors of Peer

1.1 The APIs of Peer

Since we have decided to employ the distributed component, Peer, let us take a look at its APIs. Table 6.1 and Table 6.2 list the primary APIs of Peer. Compared with those of CSServer in Table 5.1 and Table 5.2 of Chapter 5, the constructor of Peer becomes more complicated. In addition to the ones for CSServer, three methods are added, including asyncNotify(), syncNotify() and read(), which interact with remote nodes actively. Moreover, there are some other APIs that are usually used in the case of multicasting.

Methods	Explanation
void asyncNotify(String ip, int port, ServerMessage notification)	Send an asynchronous notification to a remote node
FreeClientPool getClientPool()	Expose the internal TCP client pool
String getLocalIPKey()	Expose the key created by its IP
String getPeerIP()	Expose the IP of the peer
ServerMessage read(String ip, int port, ServerMessage request)	Send a request and wait until the response is received
void start()	Start the peer
void stop()	Stop the peer
void syncNotify(String ip, int port, ServerMessage notification)	Send a synchronous notification to a remote node

Table 6.2 The methods of Peer

1.2 The Registry Server

Now you have learned the APIs of Peer. It seems that you are ready for programming with it. If you have sufficient knowledge about distributed systems, you can start right now. If not, you ought to be patient for a while and read the below introduction to the peer based distributed architecture.

1.2.1 Why Do We Need a Registry Server?

It is true that each peer is able to interact with each other actively with those APIs, such as read(), asyncNotify() and syncNotify(), listed in Table 6.2. However, have you noted that the approach to invoke those APIs, i.e., what are the parameters of those methods? That is, you have to specify the remote IP before starting to program with them. If the scale of the system is small, that is not a problem. You can memorize the IPs of your potential partners in a certain way. However, as we mention before, the scale of the P2P system is usually large because of its decentralized property. If so, how can you retain the large number of potential IPs? Even worse, how can you know those IPs before remembering them? Another question is why we do not consider the problem when programming with CSServer.

All of questions force us to establish a specific component before we program the chatting system. In Chapter 5, each chatting client do not need to interact with each other directly. Instead, all of them send their polling requests and notifications to the chatting server. That means the chatting system can start to work if the IP address of the chatting server is remembered by each of the chatting clients. However, in the peer-based chatting system, since each of the chatting peers need to interact with one another directly without the support of the chatting server, which plays the role of a broker, it is necessary for each of them to retrieve the IP addresses of their potential

chatting partners. To achieve this goal, a special server is set up. Its mission is to retain all of the IP addresses of the peers in the chatting system. When a new peer signs in, it is mandatory for it to submit its IP address to the server. The server keeps all of the IP addresses and it is responsible for inquiries from those peers when they need to interact with others.

In short, the registry server is the most important component of the chatting system. Without it, the chatting service can never be performed. To be more general, any distributed systems should have a registry server before they start to work. A distributed system contains more than one independent computing resources that are under the control of each individual distributed node. To make them work together, the registry server is the yellow page that each of them can locate each other. That is the first precondition to establish the system.

You must have a question about the chatting system we program in Chapter 5. Why does that system have no such a registry server? Actually, the C/S chatting system has a registry service as well. The chatting server provides chatting clients with such a service. In other words, each chatting client locates any of its potential partners through retrieving the registry on the chatting server. Do you remember the code, List 5.3 and List 5.4, in Chapter 5? Those classes retain the signed-in chatting clients such that they can find each other from them. Therefore, you must understand the chatting server has to take two important tasks, locating chatting partners and keeping chatting messages. For the special structure of C/S, it is feasible to provide a registry service implicitly. But in the peer-based system, an explicit server is an indispensible component that should be set up explicitly for the registry service.

1.2.2 What is the Distributed Component of the Registry Server?

Now let us program the registry server. As a distributed component, which one is suitable it, the client, the server, the peer or the cluster? First of all, the client cannot be the candidate since it cannot receive non-response messages. Then, what about the other three ones? Basically, each of the rest ones is a feasible choice. However, to simplify the design, a server is an appropriate choice for the registry server. Although a peer can take over the role of a server, in the peer-based chatting system, it is unnecessary for the registry server to interact with others actively. Moreover, since the potential scale is large for a peer-based chatting system, it is essential to choose the cluster to set up the registry. But until now, I have not taught you how to program a cluster and we should move forward step by step. Thus, it is a proper choice to implement the registry server as a single node on the API of CSServer.

2. Programming the Registry Server with CSServer

If you learn the skills of programming CSServer well, it is a trivial job to implement the registry server. If not, it does not matter. According to my experiences, if one piece of knowledge is new to you, you just repeat it carefully. After a couple of time, usually three or four, you will never forget it. Let us program the registry server now through the GreatFree skills, i.e., APIs, idioms and fundamental knowledge.

2.1 SP: A Distributed System Always Starts Here

The first idiom, i.e., the coding-level design pattern, you need to recall, is the SP (Starting Point). Next, no matter what you will program, just CPR one SP since it implements nothing but launching the system. Table 6.3 presents the pattern of SP for the register server. You are suggested to compare with it with the one of Table 5.1 for the code StartChatServer in List 5.1. The two patterns look identical in terms of their structures, right?

Pattern: SP (Starting Point)		
Class	StartRegistryServer	
Singleton	Yes	
Method	void main(String[] args)	
Enclosed Pattern	MS (Main Server)	RegistryServer
Employed API	TerminalSignal	

Table 6.3 The pattern of SP for the register server

List 6.1 illustrates the code of StartRegistryServer.java.

```
1   package com.greatfree.chat.registry;
2
3   import java.io.IOException;
4
5   import com.greatfree.exceptions.RemoteReadException;
6   import com.greatfree.testing.data.ServerConfig;
7   import com.greatfree.util.TerminateSignal;
8
9   // Created: 05/01/2017, Bing Li
10  public class StartRegistryServer
11  {
12      public static void main(String[] args)
13      {
14          System.out.println("Chatting registry server starting up ...");
15          try
16          {
17              RegistryServer.PEER().start();
18          }
19          catch (IOException | ClassNotFoundException | RemoteReadException e)
20          {
21              e.printStackTrace();
22          }
23          System.out.println("Chatting registry server started ...");
24
25          // After the server is started, the loop check whether the flag of terminating is set. If the terminating flag
26          // is true, the process is ended. Otherwise, the process keeps running. 08/22/2014, Bing Li
27          while (!TerminateSignal.SIGNAL().isTerminated())
28          {
29              try
30              {
31                  // If the terminating flag is false, it is required to sleep for some time. Otherwise, it might cause the
32                  // high CPU usage. 08/22/2014, Bing Li
33                  Thread.sleep(ServerConfig.TERMINATE_SLEEP);
34              }
35              catch (InterruptedException e)
36              {
37                  e.printStackTrace();
38              }
39          }
40      }
41  }
```

List 6.1 The code of StartRegistryServer.java

2.2 MS: The Main Server of the Registry

After the code to launch the system is done, the code to be programmed is the one implemented by the pattern of MS (Main Server). Before programming the code, can you recall what stuffs should be included in the pattern? Yes, the primary API in MS is the distributed component. In this case, that is CSServer. We have discussed the pattern of MS and the CSServer in Chapter 5. Thus, we do not need to explain the code in detail. List 6.2 presents the code of RegistryServer.java. Table 6.4 presents the pattern of MS in a more concise manner.

```
1    package com.greatfree.chat.registry;
2
3    import java.io.IOException;
4
5    import com.greatfree.chat.ChatConfig;
6    import com.greatfree.exceptions.RemoteReadException;
7    import com.greatfree.server.CSServer;
8    import com.greatfree.server.PeerRegistry;
9    import com.greatfree.server.PeerRegistryDispatcher;
10   import com.greatfree.testing.data.ServerConfig;
11   import com.greatfree.util.TerminateSignal;
12   import com.greatfree.util.Tools;
13   import com.greatfree.util.UtilConfig;
14
15   /*
16    * The server is a CSServer rather than a Peer since it is not necessary to interact with any distributed nodes
17    * actively but passively. Chatting peers register to the server for them to access to form the chatting
18    * sessions. 04/30/2017, Bing Li
19    */
20
21   // Created: 04/30/2017, Bing Li
22   public class RegistryServer
23   {
24       private CSServer<PeerRegistryDispatcher> peerRegistryServer;
25       private CSServer<ChatRegistryDispatcher> chatRegistryServer;
26       private CSServer<ChatManDispatcher> manServer;
27
28       private RegistryServer()
29       {
30       }
31
32       private static RegistryServer instance = new RegistryServer();
33
34       public static RegistryServer PEER()
35       {
36           if (instance == null)
37           {
38               instance = new RegistryServer();
39               return instance;
40           }
41           else
42           {
43               return instance;
44           }
45       }
46
47       public void stop() throws IOException, InterruptedException, ClassNotFoundException,
48           RemoteReadException
49       {
50           // Set the terminating signal. 11/25/2014, Bing Li
51           TerminateSignal.SIGNAL().setTerminated();
52
53           // Dispose the system-level registry. 05/01/2017, Bing Li
54           PeerRegistry.SYSTEM().dispose();
55           // Dispose the account registry. 04/30/2017, Bing Li
56           AccountRegistry.APPLICATION().dispose();
57
58           // Stop the two servers. 04/30/2017, Bing Li
59           this.peerRegistryServer.stop();
60           this.chatRegistryServer.stop();
61           this.manServer.stop();
```

```
62          }
63
64          public void start() throws IOException, ClassNotFoundException, RemoteReadException
65          {
66              // Initialize the peer registry server. 04/30/2017, Bing Li
67              this.peerRegistryServer = new CSServer.CSServerBuilder<PeerRegistryDispatcher>()
68                  .port(UtilConfig.PEER_REGISTRY_PORT)
69                  .listenerCount(ServerConfig.LISTENING_THREAD_COUNT)
70                  .listenerThreadPoolSize(ServerConfig.SHARED_THREAD_POOL_SIZE)
71                  .listenerThreadKeepAliveTime
72                      (ServerConfig.SHARED_THREAD_POOL_KEEP_ALIVE_TIME)
73                  .dispatcher(new PeerRegistryDispatcher(ServerConfig.DISPATCHER_POOL_SIZE,
74                      ServerConfig.DISPATCHER_POOL_THREAD_POOL_ALIVE_TIME,
75                      ServerConfig.SCHEDULER_POOL_SIZE,
76                      ServerConfig.SCHEDULER_KEEP_ALIVE_TIME))
77                  .build();
78
79              // Initialize the chat registry server. 04/30/2017, Bing Li
80              this.chatRegistryServer = new CSServer.CSServerBuilder<ChatRegistryDispatcher>()
81                  .port(ChatConfig.CHAT_REGISTRY_PORT)
82                  .listenerCount(ServerConfig.LISTENING_THREAD_COUNT)
83                  .listenerThreadPoolSize(ServerConfig.SHARED_THREAD_POOL_SIZE)
84                  .listenerThreadKeepAliveTime
85                      (ServerConfig.SHARED_THREAD_POOL_KEEP_ALIVE_TIME)
86                  .dispatcher(new ChatRegistryDispatcher(ChatConfig.DISPATCHER_THREAD_POOL_SIZE,
87                      ChatConfig.DISPATCHER_THREAD_POOL_KEEP_ALIVE_TIME,
88                      ChatConfig.SCHEDULER_THREAD_POOL_SIZE,
89                      ChatConfig.SCHEDULER_THREAD_POOL_KEEP_ALIVE_TIME))
90                  .build();
91
92              // Initialize the chat management server. 04/30/2017, Bing Li
93              this.manServer = new CSServer.CSServerBuilder<ChatManDispatcher>()
94                  .port(ChatConfig.CHAT_ADMIN_PORT)
95                  .listenerCount(ServerConfig.SINGLE_THREAD_COUNT)
96                  .listenerThreadPoolSize(ServerConfig.SHARED_THREAD_POOL_SIZE)
97                  .listenerThreadKeepAliveTime
98                      (ServerConfig.SHARED_THREAD_POOL_KEEP_ALIVE_TIME)
99                  .dispatcher(new ChatManDispatcher(ChatConfig.DISPATCHER_THREAD_POOL_SIZE,
100                     ChatConfig.DISPATCHER_THREAD_POOL_KEEP_ALIVE_TIME,
101                     ChatConfig.SCHEDULER_THREAD_POOL_SIZE,
102                     ChatConfig.SCHEDULER_THREAD_POOL_KEEP_ALIVE_TIME))
103                 .build();
104
105             // Register the ports to avoid potential conflicts. 05/02/2017, Bing Li
106             String localIP = Tools.getLocalIP();
107             // Register the main port. 05/02/2017, Bing Li
108             PeerRegistry.SYSTEM().register(ChatConfig.CHAT_REGISTRY_SERVER_KEY,
109                 ChatConfig.CHAT_REGISTRY_NAME, localIP, UtilConfig.PEER_REGISTRY_PORT);
110             // Register the chat registry port. 05/02/2017, Bing Li
111             PeerRegistry.SYSTEM().registeOthers(ChatConfig.CHAT_REGISTRY_SERVER_KEY,
112                 ChatConfig.CHAT_REGISTRY_PORT_KEY, localIP,
113                 ChatConfig.CHAT_REGISTRY_PORT);
114             // Register the registry administration port. 05/02/2017, Bing Li
115             PeerRegistry.SYSTEM().registeOthers(ChatConfig.CHAT_REGISTRY_SERVER_KEY,
116                 ChatConfig.REGISTRY_ADMIN_PORT_KEY, localIP, ChatConfig.CHAT_ADMIN_PORT);
117
118             // Start up the two servers. 04/30/2017, Bing Li
119             this.peerRegistryServer.start();
120             this.chatRegistryServer.start();
121             this.manServer.start();
122         }
123     }
```

List 6.2 The code of RegistryServer.java

Line 24~26 declares three instances of CSServer. Multiple instances of CSServer should be declared when your server need to perform their functions by multiple ports. Each port is mapped to one instance of CSServer. Although our system is not complicated, to present the skills of GreatFree programming, we always declare multiple instances of CSServer.

In our current case, the first instance, peerRegistryServer, of CSServer provides registry services on the system level, which is independent of the upper level application. Simply speaking, any peer-based systems should register something through the instance of CSServer. You can see the dispatcher of the CSServer is called PeerRegistryDispatcher (List 6.6), which is named in the way that is isolated from the application level, for example, the chatting.

Pattern: MS (Main Server)		
Class	RegistryServer	
Singleton	Yes	
Method	void start()	
	void stop()	
Enclosed Pattern	SD (Server Dispatcher)	CSServer<PeerRegistryDispatcher>
		CSServer<ChatRegistryDispatcher>
		CSServer<ChatManDispatcher>
Employed API	CSServer<Dispatcher extends ServerDispatcher>	

Table 6.4 The pattern of MS for the registry server

The second instance of CSServer, chatRegistryServer, is designed for receiving registry information about the chatting service, i.e., the upper level application, for example, the chatting users' preferences and so forth.

The third instance of CSServer, manServer, should not be strange to you. We have seen a similar one previously, i.e., Line 21 of List 5.2 in Chapter 5. To be more practical, the server receives remote management instructions from an administrator.

Line 28~45 defines the registry server as a singleton.

Line 47~62 stops the registry server. The three instances of CSServer are stopped in Line 59~61. In addition, two relevant registries are disposed, including the system level registry, PeerRegistry (List 6.3), and the application level one, AccountRegistry (List 6.5), in Line 54 and Line 56, respectively.

Line 64~122 starts up the registry server.

Line 67~77 initializes the peer registry server, i.e., the system level registry. To know the parameters, you can refer to Table 5.3 and Table 5.4 in Chapter 5.

Line 80~90 initializes the account registry server, i.e., the application level registry.

Line 93~103 initializes the management server.

Line 106~116 are new lines that you have never seen. They register three IP addresses (IP and port) in the peer registry. Since the three instances work on the same machine, they share the same IP address. But they have different ports. So even though you have not opened the class of PeerRegistry (List 6.3), according to the lines, you can guess that IP addresses are the primary information that should be retained in the peer registry.

Line 106 obtains the local IP.

Line 108~109 registers the IP address which serves for the peer registry, i.e., the system level registry.

Line 111~113 registers the IP address which serves for the chatting registry, i.e., the application level registry.

Line 115~116 registers the IP address which serves for the administration.

Line 119~121 start the three servers.

2.3 The Registries

Let us take a look at the system level and application level registries. The code should be programmed by yourself. Since they are not relevant to distributed techniques, you cannot get assistance from GreatFree. The situation is changed in later chapters when you learn how to program a cluster and a distributed storage system, the simple implementation in the current section should be updated by GreatFree techniques as well. Because of that, I do not explain the code in detail. Actually, you can implement the registry in any ways you prefer.

2.3.1 The System Level Registry

List 6.3 illustrates the code of PeerRegistry.java.

```
1    package com.greatfree.server;
2
3    import java.util.HashMap;
4    import java.util.Map;
5    import java.util.Set;
6    import java.util.concurrent.ConcurrentHashMap;
7
8    import com.google.common.collect.Sets;
9    import com.greatfree.util.IPAddress;
10   import com.greatfree.util.Tools;
11   import com.greatfree.util.UtilConfig;
12
13   /*
14    * All of the signed-in peers must be retained in the registry. 05/01/2017, Bing Li
15    *
16    * As a demonstration, it is fine. For a practical system, the design is critical and tough and it
17    * should be extended. 05/01/2017, Bing Li
18    *
19    */
20
21   // Created: 05/01/2017, Bing Li
22   public class PeerRegistry
23   {
24       // Declare a map to keep all of the accounts. 11/09/2014, Bing Li
25       private Map<String, PeerAccount> accounts;
26
27       // Busy ports are retained in the collection. 05/01/2017, Bing Li
28       private Map<String, Set<Integer>> busyPorts;
29
30       /*
31        * Initialize. 11/09/2014, Bing Li
32        */
33       private PeerRegistry()
34       {
35           // Define a synchronized list for the consideration of consistency. 11/09/2014, Bing Li
36           this.accounts = new ConcurrentHashMap<String, PeerAccount>();
37           // Initialize the busy ports collection. 05/01/2017, Bing Li
38           this.busyPorts = new ConcurrentHashMap<String, Set<Integer>>();
```

```
39          }
40
41          /*
42           * A singleton implementation. 11/09/2014, Bing Li
43           */
44          private static PeerRegistry instance = new PeerRegistry();
45
46          public static PeerRegistry SYSTEM()
47          {
48              if (instance == null)
49              {
50                  instance = new PeerRegistry();
51                  return instance;
52              }
53              else
54              {
55                  return instance;
56              }
57          }
58
59          /*
60           * Dispose the registry. 11/09/2014, Bing Li
61           */
62          public void dispose()
63          {
64              this.accounts.clear();
65              this.accounts = null;
66
67              this.busyPorts.clear();
68              this.busyPorts = null;
69          }
70
71          /*
72           * Decide whether a peer is registered. 05/30/2017, Bing Li
73           */
74          public boolean isExisted(String peerKey)
75          {
76              return this.accounts.containsKey(peerKey);
77          }
78
79          /*
80           * Get one registered peer. 05/30/2017, Bing Li
81           */
82          public PeerAccount get(String peerKey)
83          {
84              if (this.accounts.containsKey(peerKey))
85              {
86                  return this.accounts.get(peerKey);
87              }
88              return UtilConfig.NO_PEER_ACCOUNT;
89          }
90
91          /*
92           * Get all of the IP addresses registered in the registry. 05/30/2017, Bing Li
93           */
94          public Map<String, IPAddress> getIPPorts()
95          {
96              Map<String, IPAddress> ips = new HashMap<String, IPAddress>();
97              for (PeerAccount entry : this.accounts.values())
98              {
99                  ips.put(entry.getPeerKey(), new IPAddress(entry.getPeerKey(), entry.getIP(), entry.getPeerPort()));
100             }
101             return ips;
102         }
103
104         /*
105          * Register one peer. 05/30/2017, Bing Li
106          */
107         public synchronized int register(String peerKey, String peerName, String ip, int port)
108         {
109             if (!this.accounts.containsKey(peerKey))
110             {
111                 String ipKey = Tools.getHash(ip);
112                 if (!this.busyPorts.containsKey(ipKey))
113                 {
114                     Set<Integer> ports = Sets.newHashSet();
115                     ports.add(port);
```

```
116                 this.busyPorts.put(ipKey, ports);
117             }
118             else
119             {
120                 port = this.getIdlePort(ipKey, port);
121             }
122             this.accounts.put(peerKey, new PeerAccount(peerKey, peerName, ip, port));
123         }
124         return port;
125     }
126
127     /*
128      * Register other ports. 05/02/2017, Bing Li
129      */
130     public synchronized void registeOthers(String peerKey, String portKey, String ip, int port)
131     {
132         this.accounts.get(peerKey).addPort(portKey, port);
133         this.busyPorts.get(Tools.getHash(ip)).add(port);
134     }
135
136     /*
137      * Unregister one peer. 05/20/2017, Bing Li
138      */
139     public synchronized void unregister(String peerKey)
140     {
141         PeerAccount account = this.accounts.get(peerKey);
142         String ipKey = Tools.getHash(account.getIP());
143         System.out.println("removed port = " + account.getPeerPort());
144         this.busyPorts.get(ipKey).remove(account.getPeerPort());
145         for (Integer port : account.getOtherPorts())
146         {
147             System.out.println("removed port = " + port);
148             this.busyPorts.get(ipKey).remove(port);
149         }
150         if (this.busyPorts.get(ipKey).isEmpty())
151         {
152             this.busyPorts.remove(ipKey);
153         }
154         this.accounts.remove(peerKey);
155     }
156
157     /*
158      * Get the idle port for the port conflicted peer. 05/02/2017, Bing Li
159      */
160     public synchronized int getIdlePort(String peerKey, String portKey, String ip, int port)
161     {
162         String ipKey = Tools.getHash(ip);
163         int idlePort = this.getIdlePort(ipKey, port);
164         this.accounts.get(peerKey).addPort(portKey, idlePort);
165         return idlePort;
166     }
167
168     /*
169      * Get the idle port for the port conflicted peer. 05/02/2017, Bing Li
170      */
171     private int getIdlePort(String ipKey, int port)
172     {
173         if (this.busyPorts.get(ipKey).contains(port))
174         {
175             port += this.busyPorts.get(ipKey).size();
176             while (this.busyPorts.get(ipKey).contains(port))
177             {
178                 port++;
179             }
180         }
181         this.busyPorts.get(ipKey).add(port);
182         return port;
183     }
184 }
```

List 6.3 The code of PeerRegistry.java

Line 25 declares a map that keeps the registered peer accounts.

Line 28 declares a map that retains all of ports, which are occupied by peers. The key of the map is the hash string generated from an IP. The reason to do that aims to avoid the possibilities of IP address conflicts on one distributed node. If only one peer runs on a single node, the design is useless. But it is possible that multiple peers run on the same distributed node. It often happens when you test the code with a very limited machines. If so, the peers on the same node should get IP conflicts if they run with the identical port. To avoid the problem, the peer registry need to detects an idle port for the peer which is registering. To find the idle one, the busy ones must be kept in the map.

Line 33~58 implements a singleton.

Line 62~69 disposes the collections.

Line 74~66 detects whether one peer is registered or not.

Line 82~89 retrieves one registered peer.

Line 94~102 gets all of IP addresses registered in the registry.

Line 107~124 registers one peer and its IP address.

Line 130~133 registers one peer and other ports it occupies.

Line 139~155 unregisters one peer. The method is executed when a peer leaves the system.

Line 160~166 gets the idle port for a peer.

Line 171~183 gets the idle port for a peer. This is a private method, which is called by two public methods, register() and getIdlePort(), in the registry.

The registry employs one class, PeerAccount, to keep the registered peer. The code is shown in List 6.4. The code encloses the key, the name, the IP and all of the ports of one registered peer.

```
1    package com.greatfree.server;
2
3    import java.util.Collection;
4    import java.util.Map;
5    import java.util.concurrent.ConcurrentHashMap;
6
7    /*
8     * The account keeps the information about signed-in peer. 05/01/2017, Bing Li
9     */
10
11   // Created: 05/01/2017, Bing Li
12   public class PeerAccount
13   {
14       private String peerKey;
15       private String peerName;
16       private String ip;
17       private int peerPort;
18       private Map<String, Integer> otherPorts;
19
20       public PeerAccount(String peerKey, String peerName, String ip, int peerPort)
21       {
22           this.peerKey = peerKey;
```

```
23          this.peerName = peerName;
24          this.ip = ip;
25          this.peerPort = peerPort;
26          this.otherPorts = new ConcurrentHashMap<String, Integer>();
27      }
28
29      public String getPeerKey()
30      {
31          return this.peerKey;
32      }
33
34      public String getPeerName()
35      {
36          return this.peerName;
37      }
38
39      public String getIP()
40      {
41          return this.ip;
42      }
43
44      public int getPeerPort()
45      {
46          return this.peerPort;
47      }
48
49      public void addPort(String key, int port)
50      {
51          this.otherPorts.put(key, port);
52      }
53
54      public int getPort(String key)
55      {
56          return this.otherPorts.get(key);
57      }
58
59      public Collection<Integer> getOtherPorts()
60      {
61          return this.otherPorts.values();
62      }
63  }
```

List 6.4 The code of PeerAccount.java

2.3.2 The Application Level Registry

Besides the system level registry, the registry server needs to retain some information related to the chatting application. The code is totally contrived since it has nothing to do with GreatFree at this moment. All of the participants of the chatting should be registered in the registry. In other words, those components, such as the registry server, which supports the distributed system, are not registered here.

```
1   package com.greatfree.chat.registry;
2
3   import java.util.Collection;
4   import java.util.Map;
5   import java.util.Set;
6   import java.util.concurrent.ConcurrentHashMap;
7
8   import com.google.common.collect.Sets;
9   import com.greatfree.util.IPAddress;
10
11  /*
12   * This is an account registry. All of the accounts of the chatting system are retained in it. 04/30/2017, Bing Li
13   */
14
15  // Created: 04/30/2017, Bing Li
16  public class AccountRegistry
17  {
18      private Map<String, PeerChatAccount> accounts;
```

```
19
20          private AccountRegistry()
21          {
22              // Define a concurrent map for the consideration of consistency. 11/09/2014, Bing Li
23              this.accounts = new ConcurrentHashMap<String, PeerChatAccount>();
24          }
25
26          /*
27           * A singleton implementation. 11/09/2014, Bing Li
28           */
29          private static AccountRegistry instance = new AccountRegistry();
30
31          public static AccountRegistry APPLICATION()
32          {
33              if (instance == null)
34              {
35                  instance = new AccountRegistry();
36                  return instance;
37              }
38              else
39              {
40                  return instance;
41              }
42          }
43
44          public void dispose()
45          {
46              this.accounts.clear();
47              this.accounts = null;
48          }
49
50          public boolean isAccountExisted(String userKey)
51          {
52              return this.accounts.containsKey(userKey);
53          }
54
55          public void add(PeerChatAccount account)
56          {
57              if (!this.accounts.containsKey(account.getUserKey()))
58              {
59                  this.accounts.put(account.getUserKey(), account);
60              }
61          }
62
63          public Collection<PeerChatAccount> getAllAccounts()
64          {
65              return this.accounts.values();
66          }
67
68          public PeerChatAccount getAccount(String key)
69          {
70              if (this.accounts.containsKey(key))
71              {
72                  return this.accounts.get(key);
73              }
74              return null;
75          }
76
77          /*
78           * The input is the IP addresses obtained from the PeerRegistry. However it must contain some IPs
79           * that do not participate the application level. So they should be excluded from the IP addresses that
80           * are related to the application. 05/30/2017, Bing Li
81           */
82          public Map<String, IPAddress> getIPPorts(Map<String, IPAddress> ipPorts)
83          {
84              Set<String> nonPeerKeys = Sets.newHashSet();
85              for (String ipEntry : ipPorts.keySet())
86              {
87                  if (!this.accounts.containsKey(ipEntry))
88                  {
89                      nonPeerKeys.add(ipEntry);
90                  }
91              }
92              for (String key : nonPeerKeys)
93              {
94                  ipPorts.remove(key);
95              }
```

```
96          return ipPorts;
97      }
98  }
```

List 6.5 The code of AccountRegistry.java

2.4 SD

As a typical server side, the pattern of SD (Server Dispatcher) always plays the crucial role of the code you need to program. You should not feel strange about the code presented in the section since you have done similar things many times in previous chapters.

2.4.1 PeerRegistryDispatcher: Registering the System Level Resources

The first SD you see in the registry server is PeerRegistryDispatcher at Line 24 of the code, RegistryServer.java, in List 6.2. Each instance of CSServer should have their own respective dispatchers. This is not new to you. Table 6.5 summarizes the pattern of SD for the peer registry. Compared with the one, CSServer<ChatServerDispatcher>, in Table 5.7, while the SD does not employ the pattern of ND, their structures are still identical. List 6.6 shows the code of PeerRegistryDispatcher.java.

Pattern: SD (Server Dispatcher)		
Class	CSServer<PeerRegistryDispatcher>	
Singleton	No	
Method	void consume(OutMessageStream<ServerMessage> message)	
	void shutdown()	
Enclosed Pattern	RD (Request Dispatcher)	RequestDispatcher<RegisterPeerRequest, RegisterPeerStream, RegisterPeerResponse, RegisterPeerThread, RegisterPeerThreadCreator>
		RequestDispatcher<PortRequest, PortStream, PortResponse, PortRequestThread, PortRequestThreadCreator>
		RequestDispatcher<ClusterIPRequest, ClusterIPStream, ClusterIPResponse, ClusterIPRequestThread, ClusterIPRequestThreadCreator>
		RequestDispatcher<UnregisterPeerRequest, UnregisterPeerStream, UnregisterPeerResponse, UnregisterPeerThread, UnregisterPeerThreadCreator>
Employed API	ServerDispatcher	
	RequestDispatcher<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage, RequestThread extends RequestQueue<Request, Stream, Response>, ThreadCreator extends RequestThreadCreatable<Request, Stream, Response, RequestThread>>	

Table 6.5 The pattern of SD for the peer registry

```
1   package com.greatfree.server;
2
3   import java.util.Calendar;
4
5   import com.greatfree.chat.registry.ClusterIPRequestThread;
6   import com.greatfree.chat.registry.ClusterIPRequestThreadCreator;
7   import com.greatfree.concurrency.RequestDispatcher;
8   import com.greatfree.message.ClusterIPRequest;
9   import com.greatfree.message.ClusterIPResponse;
10  import com.greatfree.message.ClusterIPStream;
11  import com.greatfree.message.PortRequest;
12  import com.greatfree.message.PortResponse;
13  import com.greatfree.message.PortStream;
14  import com.greatfree.message.RegisterPeerRequest;
15  import com.greatfree.message.RegisterPeerResponse;
```

```
16  import com.greatfree.message.RegisterPeerStream;
17  import com.greatfree.message.ServerMessage;
18  import com.greatfree.message.SystemMessageType;
19  import com.greatfree.message.UnregisterPeerRequest;
20  import com.greatfree.message.UnregisterPeerResponse;
21  import com.greatfree.message.UnregisterPeerStream;
22  import com.greatfree.remote.OutMessageStream;
23  import com.greatfree.testing.data.ServerConfig;
24
25  /*
26   * 1) Since a peer probably conflicts with others' ports, the registry server needs to assign idle ports to
27   * those peers.
28   *
29   * 2) Before each peer interacts with each other, they needs to get their IP address from the registry
30   * server.
31   *
32   * 05/01/2017, Bing Li
33   *
34   */
35
36  // Created: 05/01/2017, Bing Li
37  public class PeerRegistryDispatcher extends ServerDispatcher
38  {
39      // Declare a request dispatcher to respond peers' registering requests concurrently. 05/01/2017, Bing Li
40      private RequestDispatcher<RegisterPeerRequest, RegisterPeerStream, RegisterPeerResponse,
41          RegisterPeerThread, RegisterPeerThreadCreator> registerPeerRequestDispatcher;
42
43      // Declare a request dispatcher to respond a peer's idle port request concurrently. 05/02/2017, Bing Li
44      private RequestDispatcher<PortRequest, PortStream, PortResponse, PortRequestThread,
45          PortRequestThreadCreator> portRequestDispatcher;
46
47      // Declare a request dispatcher to respond IP addresses of the cluster requests
48      // concurrently. 04/17/2017, Bing Li
49      private RequestDispatcher<ClusterIPRequest, ClusterIPStream, ClusterIPResponse,
50          ClusterIPRequestThread, ClusterIPRequestThreadCreator> clusterIPRequestDispatcher;
51
52      // Declare a request dispatcher to respond peers' unregistering requests concurrently. 05/01/2017, Bing Li
53      private RequestDispatcher<UnregisterPeerRequest, UnregisterPeerStream, UnregisterPeerResponse,
54          UnregisterPeerThread, UnregisterPeerThreadCreator> unregisterPeerRequestDispatcher;
55
56      /*
57       * The constructor of PeerRegistryDispatcher. 05/01/2017, Bing Li
58       */
59      public PeerRegistryDispatcher(int threadPoolSize, long threadKeepAliveTime, int schedulerPoolSize,
69          long schedulerKeepAliveTime)
70      {
71          super(threadPoolSize, threadKeepAliveTime, schedulerPoolSize, schedulerKeepAliveTime);
72
73          this.registerPeerRequestDispatcher = new RequestDispatcher.RequestDispatcherBuilder
74              <RegisterPeerRequest, RegisterPeerStream, RegisterPeerResponse, RegisterPeerThread,
75              RegisterPeerThreadCreator>()
76              .poolSize(ServerConfig.REQUEST_DISPATCHER_POOL_SIZE)
77              .keepAliveTime(ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME)
78              .threadCreator(new RegisterPeerThreadCreator())
79              .maxTaskSize(ServerConfig.MAX_REQUEST_TASK_SIZE)
80              .dispatcherWaitTime(ServerConfig.REQUEST_DISPATCHER_WAIT_TIME)
81              .waitRound(ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND)
82              .idleCheckDelay(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY)
83              .idleCheckPeriod(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD)
84              .scheduler(super.getSchedulerPool())
85              .build();
86
87          this.portRequestDispatcher = new RequestDispatcher.RequestDispatcherBuilder<PortRequest,
88              PortStream, PortResponse, PortRequestThread, PortRequestThreadCreator>()
89              .poolSize(ServerConfig.REQUEST_DISPATCHER_POOL_SIZE)
90              .keepAliveTime(ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME)
91              .threadCreator(new PortRequestThreadCreator())
92              .maxTaskSize(ServerConfig.MAX_REQUEST_TASK_SIZE)
93              .dispatcherWaitTime(ServerConfig.REQUEST_DISPATCHER_WAIT_TIME)
94              .waitRound(ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND)
95              .idleCheckDelay(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY)
96              .idleCheckPeriod(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD)
97              .scheduler(super.getSchedulerPool())
98              .build();
99
100         this.clusterIPRequestDispatcher = new RequestDispatcher.RequestDispatcherBuilder
101             <ClusterIPRequest, ClusterIPStream, ClusterIPResponse, ClusterIPRequestThread,
```

```
102             ClusterIPRequestThreadCreator>()
103             .poolSize(ServerConfig.REQUEST_DISPATCHER_POOL_SIZE)
104             .keepAliveTime(ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME)
105             .threadCreator(new ClusterIPRequestThreadCreator())
106             .maxTaskSize(ServerConfig.MAX_REQUEST_TASK_SIZE)
107             .dispatcherWaitTime(ServerConfig.REQUEST_DISPATCHER_WAIT_TIME)
108             .waitRound(ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND)
109             .idleCheckDelay(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY)
110             .idleCheckPeriod(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD)
111             .scheduler(super.getSchedulerPool())
112             .build();
113
114         this.unregisterPeerRequestDispatcher = new RequestDispatcher.RequestDispatcherBuilder
115             <UnregisterPeerRequest, UnregisterPeerStream, UnregisterPeerResponse, UnregisterPeerThread,
116             UnregisterPeerThreadCreator>()
117             .poolSize(ServerConfig.REQUEST_DISPATCHER_POOL_SIZE)
118             .keepAliveTime(ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME)
119             .threadCreator(new UnregisterPeerThreadCreator())
120             .maxTaskSize(ServerConfig.MAX_REQUEST_TASK_SIZE)
121             .dispatcherWaitTime(ServerConfig.REQUEST_DISPATCHER_WAIT_TIME)
122             .waitRound(ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND)
123             .idleCheckDelay(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY)
124             .idleCheckPeriod(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD)
125             .scheduler(super.getSchedulerPool())
126             .build();
127     }
128
129     /*
130      * Shut down the server message dispatcher. 04/15/2017, Bing Li
131      */
132     public void shutdown() throws InterruptedException
133     {
134         this.registerPeerRequestDispatcher.dispose();
135         this.portRequestDispatcher.dispose();
137         this.clusterIPRequestDispatcher.dispose();
138         this.unregisterPeerRequestDispatcher.dispose();
139         super.shutdown();
140     }
141
142     /*
143      * Process the available messages in a concurrent way. 04/17/2017, Bing Li
144      */
145     public void consume(OutMessageStream<ServerMessage> message)
146     {
147         // Check the types of received messages. 04/17/2017, Bing Li
148         switch (message.getMessage().getType())
149         {
150             case SystemMessageType.REGISTER_PEER_REQUEST:
151                 System.out.println("REGISTER_PEER_REQUEST received @" +
152                     Calendar.getInstance().getTime());
153                 // Check whether the registry dispatcher is ready. 04/17/2017, Bing Li
154                 if (!this.registerPeerRequestDispatcher.isReady())
155                 {
156                     // Execute the registry dispatcher as a thread. 04/17/2017, Bing Li
157                     super.execute(this.registerPeerRequestDispatcher);
158                 }
159                 // Enqueue the request into the dispatcher for concurrent responding. 04/17/2017, Bing Li
160                 this.registerPeerRequestDispatcher.enqueue(new RegisterPeerStream
161                     (message.getOutStream(), message.getLock(), (RegisterPeerRequest)
162                     message.getMessage()));
163                 break;
164
165             case SystemMessageType.PORT_REQUEST:
166                 System.out.println("PORT_REQUEST received @" + Calendar.getInstance().getTime());
167                 // Check whether the port request dispatcher is ready. 04/17/2017, Bing Li
168                 if (!this.portRequestDispatcher.isReady())
169                 {
170                     // Execute the registry dispatcher as a thread. 04/17/2017, Bing Li
171                     super.execute(this.portRequestDispatcher);
172                 }
173                 // Enqueue the request into the dispatcher for concurrent responding. 04/17/2017, Bing Li
174                 this.portRequestDispatcher.enqueue(new PortStream(message.getOutStream(),
175                     message.getLock(), (PortRequest) message.getMessage()));
176                 break;
177
178             case SystemMessageType.CLUSTER_IP_REQUEST:
179                 System.out.println("CLUSTER_IP_REQUEST received @" + Calendar.getInstance().getTime());
```

```
180                 // Check whether the port request dispatcher is ready. 04/17/2017, Bing Li
181                 if (!this.clusterIPRequestDispatcher.isReady())
182                 {
183                     // Execute the registry dispatcher as a thread. 04/17/2017, Bing Li
184                     super.execute(this.clusterIPRequestDispatcher);
185                 }
186                 // Enqueue the request into the dispatcher for concurrent responding. 04/17/2017, Bing Li
187                 this.clusterIPRequestDispatcher.enqueue(new ClusterIPStream(message.getOutStream(),
188                     message.getLock(), (ClusterIPRequest) message.getMessage()));
189                 break;
190
191             case SystemMessageType.UNREGISTER_PEER_REQUEST:
192                 System.out.println("UNREGISTER_PEER_REQUEST received @" +
193                     Calendar.getInstance().getTime());
194                 // Check whether the port request dispatcher is ready. 04/17/2017, Bing Li
195                 if (!this.unregisterPeerRequestDispatcher.isReady())
196                 {
197                     // Execute the registry dispatcher as a thread. 04/17/2017, Bing Li
198                     super.execute(this.unregisterPeerRequestDispatcher);
199                 }
200                 // Enqueue the request into the dispatcher for concurrent responding. 04/17/2017, Bing Li
201                 this.unregisterPeerRequestDispatcher.enqueue(new UnregisterPeerStream
202                     (message.getOutStream(), message.getLock(), (UnregisterPeerRequest)
203                         message.getMessage()));
204                 break;
205         }
206     }
207 }
```

List 6.6 The code of PeerRegistryDispatcher.java

The SD contains four RDs (Request Dispatchers). The first one, registerPeerRequestDispatcher in Line 40~41, is responsible for registering a new incoming peer. The second one, portRequestDispatcher in Line 44~45, responds to a remote peer with an idle port that can be employed by it. The third one, clusterIPRequestDispatcher in Line 49~50, responds to a remote peer with all of the registered IP addresses. The feature is not used in the chatting system. We will know how to utilize it in the later chapter about clustering. The last one, unregisterPeerRequestDispatcher in Line 53~54, starts to work when a peer leaves the chatting system such that the RD needs to unregister it.

Line 59~127 is the constructor of the SD of PeerRegistryDispatcher. The constructor is called by Line 73~76 of the code, RegistryServer.java, in List 6.2. If you feel confused about the code, I suggest you review Section 2.3 of Chapter 5.

Line 132~140 shuts down the SD when necessary.

Line 145~206 dispatchers incoming requests to specific RDs such that they can be processed concurrently.

2.4.2 ChatRegistryDispatcher: Registering the Application Level Information

Besides the system level resources that must be registered, it is also necessary to register some information about chatting, i.e., the application level information. As a contrived system to teach you how to program, the design is very simple. The SD of ChatRegistryDispatcher is responsible for the task. Table 6.6 illustrates the SD and List 6.7 shows the code of ChatRegistryDispatcher.java. Compared with the SD in Table 5.7, the pattern of SD in this case is also simple for the lack of ND. However,

their structures are identical such that you can program with SD when dispatching messages at the server side.

```
1   package com.greatfree.chat.registry;
2
3   import java.util.Calendar;
4
5   import com.greatfree.chat.message.ChatMessageType;
6   import com.greatfree.chat.message.peer.ChatPartnerRequest;
7   import com.greatfree.chat.message.peer.ChatPartnerResponse;
8   import com.greatfree.chat.message.peer.ChatPartnerStream;
9   import com.greatfree.chat.message.peer.ChatRegistryRequest;
10  import com.greatfree.chat.message.peer.ChatRegistryResponse;
11  import com.greatfree.chat.message.peer.ChatRegistryStream;
12  import com.greatfree.concurrency.RequestDispatcher;
13  import com.greatfree.message.ServerMessage;
14  import com.greatfree.remote.OutMessageStream;
15  import com.greatfree.server.ServerDispatcher;
16  import com.greatfree.testing.data.ServerConfig;
17
18  /*
19   * The server dispatcher is responsible for receiving chatting registry messages and responding
20   * chatting requests. 04/30/2017, Bing Li
21   */
22
23  // Created: 04/30/2017, Bing Li
24  public class ChatRegistryDispatcher extends ServerDispatcher
25  {
26      // Declare a request dispatcher to respond users' registry requests concurrently. 04/17/2017, Bing Li
27      private RequestDispatcher<ChatRegistryRequest, ChatRegistryStream, ChatRegistryResponse,
28          ChatRegistryThread, ChatRegistryThreadCreator> registryRequestDispatcher;
29      // Declare a request dispatcher to respond users' chat partner requests concurrently. 04/17/2017, Bing Li
30      private RequestDispatcher<ChatPartnerRequest, ChatPartnerStream, ChatPartnerResponse,
31          ChatPartnerRequestThread, ChatPartnerRequestThreadCreator> chatPartnerRequestDispatcher;
32
33      /*
34       * The constructor of RegistryDispatcher. 04/30/2017, Bing Li
35       */
36      public ChatRegistryDispatcher(int threadPoolSize, long threadKeepAliveTime, int schedulerPoolSize,
37          long schedulerKeepAliveTime)
38      {
39          super(threadPoolSize, threadKeepAliveTime, schedulerPoolSize, schedulerKeepAliveTime);
40
41          this.registryRequestDispatcher = new RequestDispatcher.RequestDispatcherBuilder
42              <ChatRegistryRequest, ChatRegistryStream, ChatRegistryResponse, ChatRegistryThread,
43              ChatRegistryThreadCreator>()
44              .poolSize(ServerConfig.REQUEST_DISPATCHER_POOL_SIZE)
45              .keepAliveTime(ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME)
46              .threadCreator(new ChatRegistryThreadCreator())
47              .maxTaskSize(ServerConfig.MAX_REQUEST_TASK_SIZE)
48              .dispatcherWaitTime(ServerConfig.REQUEST_DISPATCHER_WAIT_TIME)
49              .waitRound(ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND)
50              .idleCheckDelay(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY)
51              .idleCheckPeriod(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD)
52              .scheduler(super.getSchedulerPool())
53              .build();
54
55          this.chatPartnerRequestDispatcher = new RequestDispatcher.RequestDispatcherBuilder
56              <ChatPartnerRequest, ChatPartnerStream, ChatPartnerResponse, ChatPartnerRequestThread,
57              ChatPartnerRequestThreadCreator>()
58              .poolSize(ServerConfig.REQUEST_DISPATCHER_POOL_SIZE)
59              .keepAliveTime(ServerConfig.REQUEST_DISPATCHER_THREAD_ALIVE_TIME)
60              .threadCreator(new ChatPartnerRequestThreadCreator())
61              .maxTaskSize(ServerConfig.MAX_REQUEST_TASK_SIZE)
62              .dispatcherWaitTime(ServerConfig.REQUEST_DISPATCHER_WAIT_TIME)
63              .waitRound(ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND)
64              .idleCheckDelay(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY)
65              .idleCheckPeriod(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD)
66              .scheduler(super.getSchedulerPool())
67              .build();
68      }
69
70      /*
71       * Shut down the server message dispatcher. 04/30/2017, Bing Li
72       */
```

```
73          public void shutdown() throws InterruptedException
74          {
75              this.registryRequestDispatcher.dispose();
76              this.chatPartnerRequestDispatcher.dispose();
77              super.shutdown();
78          }
79
80          /*
81           * Process the available messages in a concurrent way. 04/30/2017, Bing Li
82           */
83          public void consume(OutMessageStream<ServerMessage> message)
84          {
85              // Check the types of received messages. 04/30/2017, Bing Li
86              switch (message.getMessage().getType())
87              {
88                  case ChatMessageType.PEER_CHAT_REGISTRY_REQUEST:
89                      System.out.println("PEER_CHAT_REGISTRY_REQUEST received @" +
90                          Calendar.getInstance().getTime());
91                      // Check whether the registry dispatcher is ready. 04/17/2017, Bing Li
92                      if (!this.registryRequestDispatcher.isReady())
93                      {
94                          // Execute the registry dispatcher as a thread. 04/17/2017, Bing Li
95                          super.execute(this.registryRequestDispatcher);
96                      }
97                      // Enqueue the request into the dispatcher for concurrent responding. 04/17/2017, Bing Li
98                      this.registryRequestDispatcher.enqueue(new ChatRegistryStream(message.getOutStream(),
99                          message.getLock(), (ChatRegistryRequest)message.getMessage()));
100                     break;
101
102                 case ChatMessageType.PEER_CHAT_PARTNER_REQUEST:
103                     System.out.println("PEER_CHAT_PARTNER_REQUEST received @" +
104                         Calendar.getInstance().getTime());
105                     // Check whether the partner request dispatcher is ready. 04/17/2017, Bing Li
106                     if (!this.chatPartnerRequestDispatcher.isReady())
107                     {
108                         // Execute the partner request dispatcher as a thread. 04/17/2017, Bing Li
109                         super.execute(this.chatPartnerRequestDispatcher);
110                     }
111                     // Enqueue the request into the dispatcher for concurrent responding. 04/17/2017, Bing Li
112                     this.chatPartnerRequestDispatcher.enqueue(new ChatPartnerStream
113                         (message.getOutStream(), message.getLock(), (ChatPartnerRequest)
114                         message.getMessage()));
115                     break;
116             }
117         }
118     }
```

List 6.7 The code of ChatRegistryDispatcher.java

Pattern: SD (Server Dispatcher)		
Class	CSServer<ChatRegistryDispatcher>	
Singleton	No	
Method	void consume(OutMessageStream<ServerMessage> message)	
	void shutdown()	
Enclosed Pattern	RD (Request Dispatcher)	RequestDispatcher<ChatRegistryRequest, ChatRegistryStream, ChatRegistryResponse, ChatRegistryThread, ChatRegistryThreadCreator>
		RequestDispatcher<ChatPartnerRequest, ChatPartnerStream, ChatPartnerResponse, ChatPartnerRequestThread, ChatPartnerRequestThreadCreator>
Employed API	ServerDispatcher	
	RequestDispatcher<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage, RequestThread extends RequestQueue<Request, Stream, Response>, ThreadCreator extends RequestThreadCreatable<Request, Stream, Response, RequestThread>>	

Table 6.6 The pattern of SD for the chatting registry

This SD is made up with two RDs, registryRequestDispatcher and chatPartnerRequestDispatcher. The first one in Line 27~28 is used to register chatting related information and the second one in Line 30~31 answers the query about one potential chatting partner.

Line 36~68 presents the constructor of the SD, ChatRegistryDispatcher. It is called by Line 86~89 in the code of RegistryServer.java in List 6.2.

Line 73~78 shuts down the dispatcher when the system is closed.

Line 83~117 dispatches incoming requests to specific RDs to process them concurrently.

2.4.3 ChatManDispatcher: Managing the Registry Server

The last SD, ChatManDispatcher, is very simple since it contains only one ND (Notification Dispatcher). The ND receives one notification to shut sown the registry server. Table 6.7 gives a summarization of the SD pattern for the chatting administration. It is more simple since only one ND is employed. The code is shown in List 6.8.

Pattern: SD (Server Dispatcher)		
Class	CSServer<ChatManDispatcher>	
Singleton	No	
Method	void consume(OutMessageStream<ServerMessage> message)	
	void shutdown()	
Enclosed Pattern	ND (Notification Dispatcher)	NotificationDispatcher<ShutdownChatServerNotification, ShutdownChattingRegistryServerThread, ShutdownChattingRegistryServerThreadCreator>
Employed API	ServerDispatcher	
	NotificationDispatcher<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>, ThreadCreator extends NotificationThreadCreatable<Notification, NotificationThread>>	

Table 6.7 The pattern of SD for the chatting administration

```
1   package com.greatfree.chat.registry;
2
3   import java.util.Calendar;
4
5   import com.greatfree.chat.message.ChatMessageType;
6   import com.greatfree.chat.message.ShutdownChatServerNotification;
7   import com.greatfree.concurrency.NotificationDispatcher;
8   import com.greatfree.message.ServerMessage;
9   import com.greatfree.remote.OutMessageStream;
10  import com.greatfree.server.ServerDispatcher;
11  import com.greatfree.testing.data.ServerConfig;
12
13  /*
14   * The server dispatcher extends com.greatfree.server.ServerDispatcher to implement a sub server
15   * dispatcher for administration. 04/30/2017, Bing Li
16   */
17
18  // Created: 04/30/2017, Bing Li
19  public class ChatManDispatcher extends ServerDispatcher
20  {
21      // Declare a notification dispatcher to shutdown the server when such a notification is
22      // received. 04/18/2016, Bing Li
23      private NotificationDispatcher<ShutdownChatServerNotification,
24          ShutdownChattingRegistryServerThread, ShutdownChattingRegistryServerThreadCreator>
25          shutdownNotificationDispatcher;
26
27      public ChatManDispatcher(int threadPoolSize, long threadKeepAliveTime, int schedulerPoolSize,
28          long schedulerKeepAliveTime)
29      {
30          super(threadPoolSize, threadKeepAliveTime, schedulerPoolSize, schedulerKeepAliveTime);
31
32          this.shutdownNotificationDispatcher = new NotificationDispatcher.NotificationDispatcherBuilder
33              <ShutdownChatServerNotification, ShutdownChattingRegistryServerThread,
```

```
34                ShutdownChattingRegistryServerThreadCreator>()
35                .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
36                .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
37                .threadCreator(new ShutdownChattingRegistryServerThreadCreator())
38                .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
39                .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
40                .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
41                .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
42                .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
43                .scheduler(super.getSchedulerPool())
44                .build();
45        }
46
47        /*
48         * Shut down the server message dispatcher. 04/15/2017, Bing Li
49         */
50        public void shutdown() throws InterruptedException
51        {
52            this.shutdownNotificationDispatcher.dispose();
53            super.shutdown();
54        }
55
56        /*
57         * Process the available messages in a concurrent way. 04/17/2017, Bing Li
58         */
59        public void consume(OutMessageStream<ServerMessage> message)
60        {
61            // Check the types of received messages. 04/17/2017, Bing Li
62            switch (message.getMessage().getType())
63            {
64                case ChatMessageType.SHUTDOWN_CHAT_SERVER_NOTIFICATION:
65                    System.out.println("SHUTDOWN_CHAT_SERVER_NOTIFICATION received @" +
66                        Calendar.getInstance().getTime());
67                    // Check whether the shutdown notification dispatcher is ready or not. 02/15/2016, Bing Li
68                    if (!this.shutdownNotificationDispatcher.isReady())
69                    {
70                        // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
71                        super.execute(this.shutdownNotificationDispatcher);
72                    }
73                    // Enqueue the instance of ShutdownChatServerNotification into the dispatcher for concurrent
74                    // processing. 02/15/2016, Bing Li
75                    this.shutdownNotificationDispatcher.enqueue((ShutdownChatServerNotification)
76                        message.getMessage());
77                    break;
78            }
79        }
80    }
```

List 6.8 The code of ChatManDispatcher.java

Line 23~25 declares the RD, shutdownNotificationDispatcher.

Line 27~45 defines the constructor of the SD, ChatManDispatcher.

Line 50~54 shuts down the SD.

Line 59~79 dispatches the incoming notification, ShutdownChatServerNotification, to process it concurrently.

2.5 The System Level RD

Two of the SDs contain RDs. As discussed in the previous section, the total number of RDs employed by the chatting registry server is six. You should have been familiar with how to program them. Let us recall the five components of an RD before presenting the code. It should include the request, the output message stream, the

response, the request queue and the request queue creator. All of the RDs can be roughly divided into the system level RDs and the application level RDs. Fortunately, from the view of programming, there is no difference. This section explains the RD for the system level and all of them are enclosed in the SD of CSServer<PeerRegistryDispatcher>.

2.5.1 Registering Peers

Table 6.8 presents the pattern of RD for registering peers. The structure is identical to those RDs in Table 5.12, Table 5.13, Table 5.17 and so forth. The pattern of RD helps you a lot when dispatching requests on the server side.

Pattern: RD (Request Dispatcher)		
Class	RequestDispatcher<RegisterPeerRequest, RegisterPeerStream, RegisterPeerResponse, RegisterPeerThread, RegisterPeerThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(RegisterPeerStream request)	
Enclosed Pattern	Request	RegisterPeerRequest
	OMS (Output Message Stream)	RegisterPeerStream
	Response	RegisterPeerResponse
	RQ (Request Queue)	RegisterPeerThread
	RQC (Request Queue Creator)	RegisterPeerThreadCreator
Employed API	ServerMessage	
	OutMessageStream<Message extends ServerMessage>	
	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	RequestThreadCreatable<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage, RequestThread extends RequestQueue<Request, Stream, Response>>	

Table 6.8 The pattern of RD for processing peer registry requests

The request, RegisterPeerRequest, of the RD, registerPeerRequestDispatcher, for registering peers is shown in List 6.9. The incoming peer has to send its IP (Line 16) and port (Line 17) to the registry server. If no conflicts, the IP address can be used. If conflicts exist, the registry finds idle one for the peer.

```
1   package com.greatfree.message;
2
3   /*
4    * The request is sent from a peer to the registry server to obtain an available port on the local node. It is
5    * possible that the local node has more than one peers. If so, their ports should not
6    * conflict. 05/01/2017, Bing Li
7    */
8
9   // Created: 05/01/2017, Bing Li
10  public class RegisterPeerRequest extends ServerMessage
11  {
12      private static final long serialVersionUID = 1159700115385713220L;
13
14      private String peerKey;
15      private String peerName;
16      private String ip;
17      private int port;
18
19      public RegisterPeerRequest(String peerKey, String peerName, String ip, int port)
20      {
21          super(SystemMessageType.REGISTER_PEER_REQUEST);
22          this.peerKey = peerKey;
23          this.peerName = peerName;
24          this.ip = ip;
25          this.port = port;
26      }
```

```
27
28          public String getPeerKey()
29          {
30              return this.peerKey;
31          }
32
33          public String getPeerName()
34          {
35              return this.peerName;
36          }
37
38          public String getIP()
39          {
40              return this.ip;
41          }
42
43          public int getPort()
44          {
45              return this.port;
46          }
47      }
```

List 6.9 The code of RegisterPeerRequest.java

The output message stream, RegisterPeerStream, of the RD, registerPeerRequestDispatcher, is shown in List 6.10. The component is primarily used to enclose the output message stream that can reply to peers.

```
1       package com.greatfree.message;
2
3       import java.io.ObjectOutputStream;
4       import java.util.concurrent.locks.Lock;
5
6       import com.greatfree.remote.OutMessageStream;
7
8       /*
9        * The class encloses the output stream to send the response of RegisterPeerResponse to
10       * the peer. 05/01/2017, Bing Li
11       */
12
13      // Created: 05/01/2017, Bing Li
14      public class RegisterPeerStream extends OutMessageStream<RegisterPeerRequest>
15      {
16          public RegisterPeerStream(ObjectOutputStream out, Lock lock, RegisterPeerRequest message)
17          {
18              super(out, lock, message);
19          }
20      }
```

List 6.10 The code of RegisterPeerStream.java

The response, RegisterPeerResponse, of the RD, registerPeerRequestDispatcher, is shown in List 6.11. You know that only one value, peerPort which is the port number, is returned to peers. The port is an idle one that the peer can employ if port conflicts occur.

```
1       package com.greatfree.message;
2
3       /*
4        * The response is responded from a registry server to a peer to tell it an existing port on the local node. It is
5        * possible that the local node has more than one peers. If so, their ports should not
6        * conflict. 05/01/2017, Bing Li
7        */
8
9       // Created: 05/01/2017, Bing Li
10      public class RegisterPeerResponse extends ServerMessage
11      {
12          private static final long serialVersionUID = -7028585042815549825L;
13
```

```
14        private int peerPort;
15
16        public RegisterPeerResponse(int peerPort)
17        {
18            super(SystemMessageType.REGISTER_PEER_RESPONSE);
19            this.peerPort = peerPort;
20        }
21
22        public int getPeerPort()
23        {
24            return this.peerPort;
25        }
26    }
```

List 6.11 The code of RegisterPeerResponse.java

The request queue, RegisterPeerThread, of the RD, registerPeerRequestDispatcher, is shown in List 6.12. Exactly the same as other request queue, it invokes the method, register(), of PeerRegistry. The thread of RegisterPeerThread is always written in the pattern of DWC, as presented in Table 6.9.

```
1     package com.greatfree.server;
2
3     import java.io.IOException;
4
5     import com.greatfree.concurrency.RequestQueue;
6     import com.greatfree.message.RegisterPeerRequest;
7     import com.greatfree.message.RegisterPeerResponse;
8     import com.greatfree.message.RegisterPeerStream;
9     import com.greatfree.testing.data.ServerConfig;
10
11    /*
12     * The thread registers the peer which just signs in to the peer based distributed system and
13     * generates a response to the peer. 05/01/2017, Bing Li
14     */
15
16    // Created: 05/01/2017, Bing Li
17    public class RegisterPeerThread extends RequestQueue<RegisterPeerRequest, RegisterPeerStream,
18        RegisterPeerResponse>
19    {
20        public RegisterPeerThread(int maxTaskSize)
21        {
22            super(maxTaskSize);
23        }
24
25        @Override
26        public void run()
27        {
28            RegisterPeerStream request;
29            RegisterPeerResponse response;
30            while (!this.isShutdown())
31            {
32                while (!this.isEmpty())
33                {
34                    request = this.getRequest();
35                    // Register the peer. 06/02/2017, Bing Li
36                    response = new RegisterPeerResponse(PeerRegistry.SYSTEM().register
37                        (request.getMessage().getPeerKey(), request.getMessage().getPeerName(),
38                        request.getMessage().getIP(), request.getMessage().getPort()));
39                    try
40                    {
41                        // Respond to the peer. 06/02/2017, Bing Li
42                        this.respond(request.getOutStream(), request.getLock(), response);
43                    }
44                    catch (IOException e)
45                    {
46                        e.printStackTrace();
47                    }
48                    this.disposeMessage(request, response);
49                }
50                try
51                {
```

```
52                  // Wait for some time when the queue is empty. During the period and before the thread is killed,
53                  // some new requests might be received. If so, the thread can keep working. 02/15/2016, Bing Li
54                  this.holdOn(ServerConfig.REQUEST_THREAD_WAIT_TIME);
55              }
56              catch (InterruptedException e)
57              {
58                  e.printStackTrace();
59              }
60          }
61      }
62  }
```

List 6.12 The code of RegisterPeerThread.java

Pattern: DWC (Double While Concurrency)		
Class	RegisterPeerThread	
Singleton	No	
Method	void dispose()	
	void enqueue(RegisterPeerStream request)	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
	RegisterPeerStream getRequest()	
	void respond(ObjectOutputSteram out, Lock outLock, RegisterPeerResponse response)	
Enclosed Pattern	Request	RegisterPeerRequest
	OMS (Output Message Stream)	RegisterPeerStream
	Response	RegisterPeerResponse
Employed API	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	OutMessageStream<Message extends ServerMessage>	
	ServerMessage	

Table 6.9 The pattern of DWC for processing peer registry requests

Pattern: RD (Request Dispatcher)		
Class	RequestDispatcher<ChatRegistryRequest, ChatRegistryStream, ChatRegistryResponse, ChatRegistryThread, ChatRegistryThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(ChatRegistryStream request)	
Enclosed Pattern	Request	ChatRegistryRequest
	OMS (Output Message Stream)	ChatRegistryStream
	Response	ChatRegistryResponse
	RQ (Request Queue)	ChatRegistryThread
	RQC (Request Queue Creator)	ChatRegistryThreadCreator
Employed API	ServerMessage	
	OutMessageStream<Message extends ServerMessage>	
	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	RequestThreadCreatable<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage, RequestThread extends RequestQueue<Request, Stream, Response>>	

Table 6.10 The pattern of RD for dispatching chatting registry requests

The request queue creator, RegisterPeerThreadCreator, of the RD, registerPeerRequestDispatcher, is shown in List 6.13. It creates the instances of the request queue, RegisterPeerThread, when it is really busy to process the request, RegisterPeerRequest.

```
1   package com.greatfree.server;
2
3   import com.greatfree.concurrency.RequestThreadCreatable;
4   import com.greatfree.message.RegisterPeerRequest;
5   import com.greatfree.message.RegisterPeerResponse;
6   import com.greatfree.message.RegisterPeerStream;
7
8   /*
```

```
9      * The creator generates one instance of RegisterPeerThread. It is invoked by the thread management
10     * mechanism, RequestDispatcher. 04/15/2017, Bing Li
11     */
12
13    // Created: 05/01/2017, Bing Li
14    public class RegisterPeerThreadCreator implements RequestThreadCreatable<RegisterPeerRequest,
15        RegisterPeerStream, RegisterPeerResponse, RegisterPeerThread>
16    {
17        @Override
18        public RegisterPeerThread createRequestThreadInstance(int taskSize)
19        {
20            return new RegisterPeerThread(taskSize);
21        }
22    }
```

List 6.13 The code of RegisterPeerThreadCreator.java

2.5.2 Assigning an Idle Port

The SD of PeerRegistryDispatcher contains another RD, portRequestDispatcher, that assigns an idle port and responds one peer when the peer sends the request, PortRequest, to the register server. Table 6.10 shows the pattern of RD to dispatch idle port requests.

The request, PortRequest, is shown in List 6.14.The remote peer needs to send the port, currentPort, which it plans to use and allow the register server to determine whether it conflicts with the ones which are employed by other servers.

```
1     package com.greatfree.message;
2
3     /*
4      * Since it is possible to get port conflicts, which happen when running multiple peers on a single node, it is
5      * necessary to send the request to the registry server to get an idle port. 05/02/2017, Bing Li
6      */
7
8     // Created: 05/02/2017, Bing Li
9     public class PortRequest extends ServerMessage
10    {
11        private static final long serialVersionUID = -1112030214169918144L;
12
13        private String peerKey;
14        private String portKey;
15        private String ip;
16        private int currentPort;
17
18        public PortRequest(String peerKey, String portKey, String ip, int currentPort)
19        {
20            super(SystemMessageType.PORT_REQUEST);
21            this.peerKey = peerKey;
22            this.portKey = portKey;
23            this.ip = ip;
24            this.currentPort = currentPort;
25        }
26
27        public String getPeerKey()
28        {
29            return this.peerKey;
30        }
31
32        public String getPortKey()
33        {
34            return this.portKey;
35        }
36
37        public String getIP()
38        {
39            return this.ip;
40        }
```

```
41
42          public int getCurrentPort()
43          {
44              return this.currentPort;
45          }
46      }
```

List 6.14 The code of PortRequest.java

The output message stream, PortStream, of the RD, portRequestDispatcher, is shown in List 6.15.

```
1       package com.greatfree.message;
2
3       import java.io.ObjectOutputStream;
4       import java.util.concurrent.locks.Lock;
5
6       import com.greatfree.remote.OutMessageStream;
7
8       /*
9        * The class encloses the output stream to send the response of PortResponse to the
10       * client. 05/01/2017, Bing Li
11       */
12
13      // Created: 05/02/2017, Bing Li
14      public class PortStream extends OutMessageStream<PortRequest>
15      {
16          public PortStream(ObjectOutputStream out, Lock lock, PortRequest message)
17          {
18              super(out, lock, message);
19          }
20      }
```

List 6.15 The code of PortStream.java

The response, PortResponse, of the RD, portRequestDispatcher, is presented in List 6.16. The idle port is enclosed in the response and replied to the peer.

```
1       package com.greatfree.message;
2
3       /*
4        * Since it is possible to get port conflicts, which happen when running multiple peers on a single node, it is
5        * necessary to send the request to the registry server to get an idle port. The response encloses the idle port
6        * to the requester. 05/02/2017, Bing Li
7        */
8
9       // Created: 05/02/2017, Bing Li
10      public class PortResponse extends ServerMessage
11      {
12          private static final long serialVersionUID = -1214644579199253066L;
13
14          private int port;
15
16          public PortResponse(int port)
17          {
18              super(SystemMessageType.PORT_RESPONSE);
19              this.port = port;
20          }
21
22          public int getPort()
23          {
24              return this.port;
25          }
26      }
```

List 6.16 The code of PortResponse.java

The request queue, PortRequestThread, of the RD, portRequestDispatcher, is demonstrated in List 6.17. It invokes the method, getIdlePort() of PeerRegistry, such that an idle port can be obtained for the remote peer. Again, it utilizes the pattern of DWC, as demonstrated in Table 6.11.

```
1   package com.greatfree.server;
2
3   import java.io.IOException;
4
5   import com.greatfree.concurrency.RequestQueue;
6   import com.greatfree.message.PortRequest;
7   import com.greatfree.message.PortResponse;
8   import com.greatfree.message.PortStream;
9   import com.greatfree.testing.data.ServerConfig;
10
11  /*
12   * The thread obtains an idle port and generates a response to the peer. 05/02/2017, Bing Li
13   */
14
15  // Created: 05/02/2017, Bing Li
16  public class PortRequestThread extends RequestQueue<PortRequest, PortStream, PortResponse>
17  {
18      public PortRequestThread(int maxTaskSize)
19      {
20          super(maxTaskSize);
21      }
22
23      @Override
24      public void run()
25      {
26          PortStream request;
27          PortResponse response;
28          while (!this.isShutdown())
29          {
30              while (!this.isEmpty())
31              {
32                  request = this.getRequest();
33                  // Obtain an idle port and generate the response for the remote peer. 06/02/2017, Bing Li
34                  response = new PortResponse(PeerRegistry.SYSTEM().getIdlePort
35                      (request.getMessage().getPeerKey(), request.getMessage().getPortKey(),
36                      request.getMessage().getIP(), request.getMessage().getCurrentPort()));
37                  try
38                  {
39                      // Respond to the remote peer with the assigned idle port. 06/02/2017, Bing Li
40                      this.respond(request.getOutStream(), request.getLock(), response);
41                  }
42                  catch (IOException e)
43                  {
44                      e.printStackTrace();
45                  }
46                  this.disposeMessage(request, response);
47              }
48              try
49              {
50                  // Wait for some time when the queue is empty. During the period and before the thread is killed,
51                  // some new requests might be received. If so, the thread can keep working. 02/15/2016, Bing Li
52                  this.holdOn(ServerConfig.REQUEST_THREAD_WAIT_TIME);
53              }
54              catch (InterruptedException e)
55              {
56                  e.printStackTrace();
57              }
58          }
59      }
60  }
```

List 6.17 The code of PortRequestThread.java

The request queue creator, PortRequestThreadCreator, of the RD, portRequestDispatcher, is shown in List 6.18.

```
1    package com.greatfree.server;
2
3    import com.greatfree.concurrency.RequestThreadCreatable;
4    import com.greatfree.message.PortRequest;
5    import com.greatfree.message.PortResponse;
6    import com.greatfree.message.PortStream;
7
8    // Created: 05/02/2017, Bing Li
9    public class PortRequestThreadCreator implements RequestThreadCreatable<PortRequest,
10       PortStream, PortResponse, PortRequestThread>
11   {
12       @Override
13       public PortRequestThread createRequestThreadInstance(int taskSize)
14       {
15           return new PortRequestThread(taskSize);
16       }
17   }
```

List 6.18 The code of PortRequestThreadCreator.java

Pattern: DWC (Double While Concurrency)		
Class	PortRequestThread	
Singleton	No	
Method	void dispose()	
	void enqueue(PortStream request)	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
	PortStream getRequest()	
	void respond(ObjectOutputStream out, Lock outLock, PortResponse response)	
Enclosed Pattern	Request	PortRequest
	OMS (Output Message Stream)	PortStream
	Response	PortResponse
Employed API	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	OutMessageStream<Message extends ServerMessage>	
	ServerMessage	

Table 6.11 The pattern of DWC for processing idle port requests

2.5.3 Responding All of Registered IP Addresses

The third RD of the SD is the one to dispatch the request of ClusterIPRequest, which is sent by the root of a cluster. We will discuss cluster programming later. Now we just show you the structure of the code and forget about the sense of the request temporarily at this moment. Table 6.12 shows the RD.

Let us first take a look at the request, ClusterIPRequest, for the RD. The code is shown in List 6.19. Since the request needs all of the registered IP addresses, it does not contain any properties in it.

```
1    package com.greatfree.message;
2
3    /*
4     * The request is sent to the Registry Server of the peer based system to retrieve all of registered IP
5     * addresses of the cluster. 05/08/2017, Bing Li
6     */
7
8    // Created: 05/08/2017, Bing Li
9    public class ClusterIPRequest extends ServerMessage
10   {
11       private static final long serialVersionUID = 49780739482646513L;
12
13       public ClusterIPRequest()
14       {
15           super(SystemMessageType.CLUSTER_IP_REQUEST);
```

```
16        }
17    }
```

List 6.19 The code of ClusterIPRequest.java

Pattern: RD (Request Dispatcher)		
Class	RequestDispatcher<ClusterIPRequest, ClusterIPStream, ClusterIPResponse, ClusterIPRequestThread, ClusterIPRequestThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(ClusterIPStream request)	
Enclosed Pattern	Request	ClusterIPRequest
	OMS (Output Message Stream)	ClusterIPStream
	Response	ClusterIPResponse
	RQ (Request Queue)	ClusterIPRequestThread
	RQC (Request Queue Creator)	ClusterIPRequestThreadCreator
Employed API	ServerMessage	
	OutMessageStream<Message extends ServerMessage>	
	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	RequestThreadCreatable<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage, RequestThread extends RequestQueue<Request, Stream, Response>>	

Table 6.12 The pattern of RD for dispatching cluster IP requests

The output message stream, ClusterIPStream, of the RD is shown in List 6.20.

```
1     package com.greatfree.message;
2
3     import java.io.ObjectOutputStream;
4     import java.util.concurrent.locks.Lock;
5
6     import com.greatfree.remote.OutMessageStream;
7
8     /*
9      * The class encloses the output stream to send the response of ClusterIPResponse to
10     * the client. 04/30/2017, Bing Li
11     */
12
13    // Created: 05/08/2017, Bing Li
14    public class ClusterIPStream extends OutMessageStream<ClusterIPRequest>
15    {
16        public ClusterIPStream(ObjectOutputStream out, Lock lock, ClusterIPRequest message)
17        {
18            super(out, lock, message);
19        }
20    }
```

List 6.20 The code of ClusterIPStream.java

The response, ClusterIPResponse, is presented in List 6.21. It contains a collection of map, which encloses all of the IP addresses of registered peers.

```
1     package com.greatfree.message;
2
3     import java.util.Map;
4
5     import com.greatfree.util.IPAddress;
6
7     /*
8      * All of registered IP addresses are enclosed in the message. 05/08/2017, Bing Li
9      */
10
11    // Created: 05/08/2017, Bing Li
12    public class ClusterIPResponse extends ServerMessage
13    {
14        private static final long serialVersionUID = -7503167324503868071L;
```

```
15
16          private Map<String, IPAddress> ips;
17
18          public ClusterIPResponse(Map<String, IPAddress> ips)
19          {
20              super(SystemMessageType.CLUSTER_IP_RESPONSE);
21              this.ips = ips;
22          }
23
24          public Map<String, IPAddress> getIPs()
25          {
26              return this.ips;
27          }
28      }
```

List 6.21 The code of ClusterIPResponse.java

The request queue, ClusterIPRequestThread, is shown in List 6.22. The same as others, it is implemented in the pattern of DWC. Table 6.13 lists the most components in the pattern.

```
1       package com.greatfree.chat.registry;
2
3       import java.io.IOException;
4
5       import com.greatfree.concurrency.RequestQueue;
6       import com.greatfree.message.ClusterIPRequest;
7       import com.greatfree.message.ClusterIPResponse;
8       import com.greatfree.message.ClusterIPStream;
9       import com.greatfree.server.PeerRegistry;
10      import com.greatfree.testing.data.ServerConfig;
11
12      /*
13       * The thread retrieves all of the registered IPs of the distributed nodes within one cluster. 05/08/2017, Bing Li
14       */
15
16      // Created: 05/08/2017, Bing Li
17      public class ClusterIPRequestThread extends RequestQueue<ClusterIPRequest, ClusterIPStream,
18          ClusterIPResponse>
19      {
20
21          public ClusterIPRequestThread(int maxTaskSize)
22          {
23              super(maxTaskSize);
24          }
25
26          @Override
27          public void run()
28          {
29              ClusterIPStream request;
30              ClusterIPResponse response;
31              while (!this.isShutdown())
32              {
33                  while (!this.isEmpty())
34                  {
35                      request = this.getRequest();
36                      // Generate the response that encloses all of IPs of the nodes in the cluster. 05/08/2017, Bing Li
37                      response = new ClusterIPResponse(AccountRegistry.APPLICATION().getIPPorts
38                          (PeerRegistry.SYSTEM().getIPPorts()));
39                      try
40                      {
41                          this.respond(request.getOutStream(), request.getLock(), response);
42                      }
43                      catch (IOException e)
44                      {
45                          e.printStackTrace();
46                      }
47                      this.disposeMessage(request, response);
48                  }
49                  try
50                  {
51                      // Wait for some time when the queue is empty. During the period and before the thread is killed,
52                      // some new requests might be received. If so, the thread can keep working. 02/15/2016, Bing Li
```

```
53                  this.holdOn(ServerConfig.REQUEST_THREAD_WAIT_TIME);
54              }
55              catch (InterruptedException e)
56              {
57                  e.printStackTrace();
58              }
59          }
60      }
61  }
```

List 6.22 The code of ClusterIPRequestThread.java

Pattern: DWC (Double While Concurrency)		
Class	ClusterIPRequestThread	
Singleton	No	
Method	void dispose()	
	void enqueue(ClusterIPStream request)	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
	ClusterIPStream getRequest()	
	void respond(ObjectOutputStream out, Lock outLock, ServerMessage response)	
Enclosed Pattern	Request	ClusterIPRequest
	OMS (Output Message Stream)	ClusterIPStream
	Response	ClusterIPResponse
Employed API	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	OutMessageStream<Message extends ServerMessage>	
	ServerMessage	

Table 6.13 The pattern of DWC for processing cluster IP requests

Pattern: RD (Request Dispatcher)		
Class	RequestDispatcher<UnregisterPeerRequest, UnregisterPeerStream, UnregisterPeerResponse, UnregisterPeerThread, UnregisterPeerThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(UnregisterPeerStream request)	
Enclosed Pattern	Request	UnregisterPeerRequest
	OMS (Output Message Stream)	UnregisterPeerStream
	Response	UnregisterPeerResponse
	RQ (Request Queue)	UnregisterPeerThread
	RQC (Request Queue Creator)	UnregisterPeerThreadCreator
Employed API	ServerMessage	
	OutMessageStream<Message extends ServerMessage>	
	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	RequestThreadCreatable<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage, RequestThread extends RequestQueue<Request, Stream, Response>>	

Table 6.14 The pattern of RD for dispatching the requests of unregistering peer

The request queue creator is presented in List 6.23.

```
1   package com.greatfree.chat.registry;
2
3   import com.greatfree.concurrency.RequestThreadCreatable;
4   import com.greatfree.message.ClusterIPRequest;
5   import com.greatfree.message.ClusterIPResponse;
6   import com.greatfree.message.ClusterIPStream;
7
8   /*
9    * The creator generates one instance of ClusterIPRequestThread. It is invoked by the thread management
10   * mechanism, RequestDispatcher. 04/30/2017, Bing Li
11   */
12
13  // Created: 05/08/2017, Bing Li
14  public class ClusterIPRequestThreadCreator implements RequestThreadCreatable<ClusterIPRequest,
```

```
15          ClusterIPStream, ClusterIPResponse, ClusterIPRequestThread>
16      {
17          @Override
18          public ClusterIPRequestThread createRequestThreadInstance(int taskSize)
19          {
20              return new ClusterIPRequestThread(taskSize);
21          }
22      }
```

List 6.23 The code of ClusterIPRequestThreadCreator.java

2.5.4 Unregistering Peers

When a peer leaves, it is essential to unregister from the registry server such that the ports it occupies can be collected for others. For that, a new request, UnregisterPeerRequest, should be processed. To do that, the pattern of RD needs to be used again. Table 6.14 illustrates the pattern.

Incidentally, it is fine to accomplish the task of unregistering a peer using a notification instead of a request.

The request, UnregisterPeerRequest, is shown in List 6.24. It contains the key of the peer which leaves.

```
1       package com.greatfree.message;
2
3       /*
4        * The request is sent from a peer to the registry server to unregister itself. It happens when the peer
5        * leaves the system. 05/01/2017, Bing Li
6        */
7
8       // Created: 05/20/2017, Bing Li
9       public class UnregisterPeerRequest extends ServerMessage
10      {
11          private static final long serialVersionUID = -7121842763115805356L;
12
13          private String peerKey;
14
15          public UnregisterPeerRequest(String peerKey)
16          {
17              super(SystemMessageType.UNREGISTER_PEER_REQUEST);
18              this.peerKey = peerKey;
19          }
20
21          public String getPeerKey()
22          {
23              return this.peerKey;
24          }
25      }
```

List 6.24 The code of UnregisterPeerRequest.java

The output message stream, UnregisterPeerStream, is presented in List 6.25.

```
1       package com.greatfree.message;
2
3       import java.io.ObjectOutputStream;
4       import java.util.concurrent.locks.Lock;
5
6       import com.greatfree.remote.OutMessageStream;
7
8       /*
9        * The class encloses the output stream to send the response of UnregisterPeerResponse to
10       * the peer. 05/20/2017, Bing Li
```

```
11      */
12
13      // Created: 05/20/2017, Bing Li
14      public class UnregisterPeerStream extends OutMessageStream<UnregisterPeerRequest>
15      {
16          public UnregisterPeerStream(ObjectOutputStream out, Lock lock, UnregisterPeerRequest message)
17          {
18              super(out, lock, message);
19          }
20      }
```

List 6.25 The code of UnregisterPeerStream.java

The response, UnregisterPeerResponse, is exhibited in List 6.26.

```
1       package com.greatfree.message;
2
3       /*
4        * The response is responded from a registry server to a peer to ensure whether it is succeeded to unregister
5        * from the registry server or not. 05/20/2017, Bing Li
6        */
7
8       // Created: 05/20/2017, Bing Li
9       public class UnregisterPeerResponse extends ServerMessage
10      {
11          private static final long serialVersionUID = 7449858374990299522L;
12
13          private boolean isSucceeded;
14
15          public UnregisterPeerResponse(boolean isSucceeded)
16          {
17              super(SystemMessageType.UNREGISTER_PEER_RESPONSE);
18              this.isSucceeded = isSucceeded;
19          }
20
21          public boolean isSucceeded()
22          {
23              return this.isSucceeded;
24          }
25      }
```

List 6.26 The code of UnregisterPeerResponse.java

The DWC pattern is used to implement the request queue to process the request. The code is illustrated in List 6.27. Table 6.15 summarizes the pattern of DWC.

```
1       package com.greatfree.server;
2
3       import java.io.IOException;
4
5       import com.greatfree.concurrency.RequestQueue;
6       import com.greatfree.message.UnregisterPeerRequest;
7       import com.greatfree.message.UnregisterPeerResponse;
8       import com.greatfree.message.UnregisterPeerStream;
9       import com.greatfree.testing.data.ServerConfig;
10
11      /*
12       * The thread unregisters the peer which just leaves the peer based distributed system and
13       * generates a response to the peer. 05/20/2017, Bing Li
14       */
15
16      // Created: 05/20/2017, Bing Li
17      public class UnregisterPeerThread extends RequestQueue<UnregisterPeerRequest,
18          UnregisterPeerStream, UnregisterPeerResponse>
19      {
20          public UnregisterPeerThread(int maxTaskSize)
21          {
22              super(maxTaskSize);
23          }
24
25          @Override
```

```
26        public void run()
27        {
28            UnregisterPeerStream request;
29            UnregisterPeerResponse response;
30            while (!this.isShutdown())
31            {
32                while (!this.isEmpty())
33                {
34                    request = this.getRequest();
35                    // Unregister the peer. 05/20/2017, Bing Li
36                    PeerRegistry.SYSTEM().unregister(request.getMessage().getPeerKey());
37                    response = new UnregisterPeerResponse(true);
38                    try
39                    {
40                        this.respond(request.getOutStream(), request.getLock(), response);
41                    }
42                    catch (IOException e)
43                    {
44                        e.printStackTrace();
45                    }
46                    this.disposeMessage(request, response);
47                }
48                try
49                {
50                    // Wait for some time when the queue is empty. During the period and before the thread is killed,
51                    // some new requests might be received. If so, the thread can keep working. 02/15/2016, Bing Li
52                    this.holdOn(ServerConfig.REQUEST_THREAD_WAIT_TIME);
53                }
54                catch (InterruptedException e)
55                {
56                    e.printStackTrace();
57                }
58            }
59        }
60    }
```

List 6.27 The code of UnregisterPeerThread.java

Pattern: DWC (Double While Concurrency)		
Class	UnregisterPeerThread	
Singleton	No	
Method	void dispose()	
	void enqueue(UnregisterPeerStream request)	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
	UnregisterPeerStream getRequest()	
	void respond(ObjectOutputStream out, Lock outLock, UnregisterPeerResponse response)	
Enclosed Pattern	Request	UnregisterPeerRequest
	OMS (Output Message Stream)	UnregisterPeerStream
	Response	UnregisterPeerResponse
Employed API	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	OutMessageStream<Message extends ServerMessage>	
	ServerMessage	

Table 6.15 The pattern of DWC to process and respond the requests of unregistering peers

The request creator, UnregisterPeerThreadCreator, is shown in List 6.28.

```
1     package com.greatfree.server;
2
3     import com.greatfree.concurrency.RequestThreadCreatable;
4     import com.greatfree.message.UnregisterPeerRequest;
5     import com.greatfree.message.UnregisterPeerResponse;
6     import com.greatfree.message.UnregisterPeerStream;
7
8     /*
9      * The creator generates one instance of UnregisterPeerThread. It is invoked by the thread management
10     * mechanism, RequestDispatcher. 04/15/2017, Bing Li
11     */
```

```
12
13    // Created: 05/20/2017, Bing Li
14    public class UnregisterPeerThreadCreator implements RequestThreadCreatable<UnregisterPeerRequest,
15        UnregisterPeerStream, UnregisterPeerResponse, UnregisterPeerThread>
16    {
17        @Override
18        public UnregisterPeerThread createRequestThreadInstance(int taskSize)
19        {
20            return new UnregisterPeerThread(taskSize);
21        }
22    }
```

List 6.28 The code of UnregisterPeerThreadCreator.java

2.6 The Application Level RD

The system level RD processes the requests that are suitable to all of the peer-based distributed systems. Now it is time for us to explain the code on the application level. For our case in the chapter, that is RD for the chatting system. Since it does not affect the infrastructure of the overall system, we do not need to spend much time on that and the implementation is also simple. You need to make it practical when programming an application in the real world. This section explains the RD for the applicatiohn level and all of them are enclosed in the SD of CSServer<ChatRegistryDispatcher>.

2.6.1 Registering Chatting Users

To dispatch the request of registering chatting users, an RD is designed. Table 6.16 summarizes the pattern.

Pattern: RD (Request Dispatcher)		
Class	RequestDispatcher<ChatRegistryRequest, ChatRegistryStream, ChatRegistryResponse, ChatRegistryThread, ChatRegistryThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(ChatRegistryStream request)	
Enclosed Pattern	Request	ChatRegistryRequest
	OMS (Output Message Stream)	ChatRegistryStream
	Response	ChatRegistryResponse
	RQ (Request Queue)	ChatRegistryThread
	RQC (Request Queue Creator)	ChatRegistryThreadCreator
Employed API	ServerMessage	
	OutMessageStream<Message extends ServerMessage>	
	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	RequestThreadCreatable<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage, RequestThread extends RequestQueue<Request, Stream, Response>>	

Table 6.16 The pattern of RD for dispatching chatting registry requests

The request, ChatRegistryRequest, is shown in List 6.29. It contains some simple contrived information as samples.

```
1     package com.greatfree.chat.message.peer;
2
3     import com.greatfree.chat.message.ChatMessageType;
4     import com.greatfree.message.ServerMessage;
5
6     /*
```

```
7      * The message encloses the data to register one account in the chatting system. 04/30/2017, Bing Li
8      */
9
10     // Created: 04/30/2017, Bing Li
11     public class ChatRegistryRequest extends ServerMessage
12     {
13         private static final long serialVersionUID = -3265857560426666644L;
14
15         private String peerKey;
16         private String peerName;
17         private String peerDescription;
18         private String preference;
19
20         public ChatRegistryRequest(String peerKey, String peerName, String peerDescription, String preference)
21         {
22             super(ChatMessageType.PEER_CHAT_REGISTRY_REQUEST);
23             this.peerKey = peerKey;
24             this.peerName = peerName;
25             this.peerDescription = peerDescription;
26             this.preference = preference;
27         }
28
29         public String getPeerKey()
30         {
31             return this.peerKey;
32         }
33
34         public String getPeerName()
35         {
36             return this.peerName;
37         }
38
39         public String getPeerDescription()
40         {
41             return this.peerDescription;
42         }
43
44         public String getPreference()
45         {
46             return this.preference;
47         }
48     }
```

List 6.29 The code of ChatRegistryRequest.java

The output message stream, ChatRegistryStream, is shown in List 6.30.

```
1      package com.greatfree.chat.message.peer;
2
3      import java.io.ObjectOutputStream;
4      import java.util.concurrent.locks.Lock;
5
6      import com.greatfree.remote.OutMessageStream;
7
8      /*
9       * The class encloses the output stream to send the response of ChatRegistryResponse to
10      * the client. 04/30/2017, Bing Li
11      */
12
13     // Created: 04/30/2017, Bing Li
14     public class ChatRegistryStream extends OutMessageStream<ChatRegistryRequest>
15     {
16         public ChatRegistryStream(ObjectOutputStream out, Lock lock, ChatRegistryRequest message)
17         {
18             super(out, lock, message);
19         }
20     }
```

List 6.30 The code of ChatRegistryStream.java

The response, ChatRegistryResponse, is shown in List 6.31.

```
1    package com.greatfree.chat.message.peer;
2
3    import com.greatfree.chat.message.ChatMessageType;
4    import com.greatfree.message.ServerMessage;
5
6    /*
7     * The response is used to respond to the client node about the result of the account
8     * registry. 04/30/2017, Bing Li
9     */
10
11   // Created: 04/30/2017, Bing Li
12   public class ChatRegistryResponse extends ServerMessage
13   {
14       private static final long serialVersionUID = -7714754232981283178L;
15
16       // The result of one account's registry. 04/15/2017, Bing Li
17       private boolean isSucceeded;
18
19       public ChatRegistryResponse(boolean isSucceeded)
20       {
21           super(ChatMessageType.PEER_CHAT_REGISTRY_RESPONSE);
22           this.isSucceeded = isSucceeded;
23       }
24
25       public boolean isSucceeded()
26       {
27           return this.isSucceeded;
28       }
29   }
```

List 6.31 The code of ChatRegistryResponse.java

The request queue, ChatRegistyThread, is presented in List 6.32, in which the pattern of DWC is employed to process and respond the requests. Table 6.17 gives a summary for the pattern.

```
1    package com.greatfree.chat.registry;
2
3    import java.io.IOException;
4
5    import com.greatfree.chat.message.peer.ChatRegistryRequest;
6    import com.greatfree.chat.message.peer.ChatRegistryResponse;
7    import com.greatfree.chat.message.peer.ChatRegistryStream;
8    import com.greatfree.concurrency.RequestQueue;
9    import com.greatfree.testing.data.ServerConfig;
10
11   /*
12    * The thread registers the chatting account concurrently. 04/30/2017, Bing Li
13    */
14
15   // Created: 04/30/2017, Bing Li
16   public class ChatRegistryThread extends RequestQueue<ChatRegistryRequest, ChatRegistryStream,
17       ChatRegistryResponse>
18   {
19       public ChatRegistryThread(int maxTaskSize)
20       {
21           super(maxTaskSize);
22       }
23
24       @Override
25       public void run()
26       {
27           ChatRegistryStream request;
28           ChatRegistryResponse response;
29           while (!this.isShutdown())
30           {
31               while (!this.isEmpty())
32               {
33                   request = this.getRequest();
34                   // Register the chatting peer. 05/01/2017, Bing Li
35                   AccountRegistry.APPLICATION().add(new PeerChatAccount(request.getMessage()
```

```
36                      .getPeerKey(), request.getMessage().getPeerName(), request.getMessage()
37                      .getPeerDescription(), request.getMessage().getPreference()));
38
39                  response = new ChatRegistryResponse(true);
40                  try
41                  {
42                      this.respond(request.getOutStream(), request.getLock(), response);
43                  }
44                  catch (IOException e)
45                  {
46                      e.printStackTrace();
47                  }
48                  this.disposeMessage(request, response);
49              }
50              try
51              {
52                  // Wait for some time when the queue is empty. During the period and before the thread is killed,
53                  // some new requests might be received. If so, the thread can keep working. 02/15/2016, Bing Li
54                  this.holdOn(ServerConfig.REQUEST_THREAD_WAIT_TIME);
55              }
56              catch (InterruptedException e)
57              {
58                  e.printStackTrace();
59              }
60          }
61      }
62  }
```

List 6.32 The code of ChatRegistryThread.java

Pattern: DWC (Double While Concurrency)		
Class	ChatRegistryThread	
Singleton	No	
Method	void dispose()	
	void enqueue(ChatRegistryStream request)	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
	ChatRegistryStream getRequest()	
	void respond(ObjectOutputStream out, Lock outLock, ChatRegistryResponse response)	
Enclosed Pattern	Request	ChatRegistryRequest
	OMS (Output Message Stream)	ChatRegistryStream
	Response	ChatRegistryResponse
Employed API	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	OutMessageStream<Message extends ServerMessage>	
	ServerMessage	

Table 6.17 The pattern of DWC for processing the chatting registry requests

The request queue creator, ChatRegistyThreadCreator, is presented in List 6.33.

```
1   package com.greatfree.chat.registry;
2
3   import com.greatfree.chat.message.peer.ChatRegistryRequest;
4   import com.greatfree.chat.message.peer.ChatRegistryResponse;
5   import com.greatfree.chat.message.peer.ChatRegistryStream;
6   import com.greatfree.concurrency.RequestThreadCreatable;
7
8   /*
9    * The creator generates one instance of ChatRegistryThread. It is invoked by the thread management
10   * mechanism, RequestDispatcher. 04/30/2017, Bing Li
11   */
12
13  // Created: 04/30/2017, Bing Li
14  public class ChatRegistryThreadCreator implements RequestThreadCreatable<ChatRegistryRequest,
15      ChatRegistryStream, ChatRegistryResponse, ChatRegistryThread>
16  {
17      @Override
18      public ChatRegistryThread createRequestThreadInstance(int taskSize)
19      {
20          return new ChatRegistryThread(taskSize);
```

```
21        }
22    }
```

List 6.33 The code of ChatRegistryThreadCreator.java

2.6.2 Searching One Chatting Partner

Another RD on the application level serves a chatting user that needs to search a potential chatting user according to its user key. Table 6.18 illustrates the pattern.

Pattern: RD (Request Dispatcher)		
Class	RequestDispatcher<ChatPartnerRequest, ChatPartnerStream, ChatPartnerResponse, ChatPartnerRequestThread, ChatPartnerRequestThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(OutMessageStream<Request> request)	
Enclosed Pattern	Request	ChatPartnerRequest
	OMS (Output Message Stream)	ChatPartnerStream
	Response	ChatPartnerResponse
	RQ (Request Queue)	ChatPartnerRequestThread
	RQC (Request Queue Creator)	ChatPartnerRequestThreadCreator
Employed API	ServerMessage	
	OutMessageStream<Message extends ServerMessage>	
	RequestQueue<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage>	
	RequestThreadCreatable<Request extends ServerMessage, Stream extends OutMessageStream<Request>, Response extends ServerMessage, RequestThread extends RequestQueue<Request, Stream, Response>>	

Table 6.18 The pattern of RD for dispatching the requests of searching potential chatting partner

The request, ChatPartnerRequest, is demonstrated in List 6.34. Only the partner key is enclosed in the request to search the potential chatting user.

```
1     package com.greatfree.chat.message.peer;
2
3     import com.greatfree.chat.message.ChatMessageType;
4     import com.greatfree.message.ServerMessage;
5
6     /*
7      * The request asks the chatting registry server to obtain the potential chatting partners. 04/16/2017, Bing Li
8      */
9
10    // Created: 04/30/2017, Bing Li
11    public class ChatPartnerRequest extends ServerMessage
12    {
13        private static final long serialVersionUID = 3239511889796307819L;
14
15        private String partnerKey;
16
17        public ChatPartnerRequest(String partnerKey)
18        {
19            super(ChatMessageType.PEER_CHAT_PARTNER_REQUEST);
20            this.partnerKey = partnerKey;
21        }
22
23        public String getPartnerKey()
24        {
25            return this.partnerKey;
26        }
27    }
```

List 6.34 The code of ChatPartnerRequest.java

The output message stream, ChatPartnerStream, is presented in List 6.35.

```
1    package com.greatfree.chat.message.peer;
2
3    import java.io.ObjectOutputStream;
4    import java.util.concurrent.locks.Lock;
5
6    import com.greatfree.remote.OutMessageStream;
7
8    /*
9     * The class encloses the output stream to send the response of ChatPartnerResponse to
10    *  the client. 04/30/2017, Bing Li
11    */
12
13   // Created: 04/30/2017, Bing Li
14   public class ChatPartnerStream extends OutMessageStream<ChatPartnerRequest>
15   {
16       public ChatPartnerStream(ObjectOutputStream out, Lock lock, ChatPartnerRequest message)
17       {
18           super(out, lock, message);
19       }
20   }
```

List 6.35 The code of ChatPartnerStream.java

The response, ChatPartnerResponse, is shown in List 6.36. It contains the information on both levels, the application one and the system level. So it needs to search the chatting user from both of the two registries, i.e., AccountRegistry and PeerRegistry. It is done by Line 39~67 of the code ChatPartnerRequestThread.java in List 6.37.

```
1    package com.greatfree.chat.message.peer;
2
3    import com.greatfree.chat.message.ChatMessageType;
4    import com.greatfree.message.ServerMessage;
5
6    /*
7     * The response contains the account description, especially the IP address for the Peer
8     * chatting system. 04/30/2017, Bing Li
9     */
10
11   // Created: 04/30/2017, Bing Li
12   public class ChatPartnerResponse extends ServerMessage
13   {
14       private static final long serialVersionUID = 8082301869328779879L;
15
16       private String userKey;
17       private String userName;
18       private String description;
19       private String preference;
20       private String ip;
21       private int port;
22
23       public ChatPartnerResponse(String userKey, String userName, String description, String preference,
24           String ip, int port)
25       {
26           super(ChatMessageType.PEER_CHAT_PARTNER_RESPONSE);
27           this.userKey = userKey;
28           this.userName = userName;
29           this.description = description;
30           this.preference = preference;
31           this.ip = ip;
32           this.port = port;
33       }
34
35       public String getUserKey()
36       {
37           return this.userKey;
38       }
39
40       public String getUserName()
41       {
```

```
42            return this.userName;
43        }
44
45        public String getDescription()
46        {
47            return this.description;
48        }
49
50        public String getPreference()
51        {
52            return this.preference;
53        }
54
55        public String getIP()
56        {
57            return this.ip;
58        }
59
60        public int getPort()
61        {
62            return this.port;
63        }
64    }
```

List 6.36 The code of ChatPartnerResponse.java

The request queue, ChatPartnerRequestThread, is shown in List 6.37. It is implemented with the pattern of DWC to process and respond the request. Table 6.19 illustrates the pattern.

```
1     package com.greatfree.chat.registry;
2
3     import java.io.IOException;
4
5     import com.greatfree.chat.message.peer.ChatPartnerRequest;
6     import com.greatfree.chat.message.peer.ChatPartnerResponse;
7     import com.greatfree.chat.message.peer.ChatPartnerStream;
8     import com.greatfree.concurrency.RequestQueue;
9     import com.greatfree.server.PeerAccount;
10    import com.greatfree.server.PeerRegistry;
11    import com.greatfree.testing.data.ServerConfig;
12    import com.greatfree.util.UtilConfig;
13
14    /*
15     * The thread checks the retrieved user of chatting. 04/16/2017, Bing Li
16     */
17
18    // Created: 04/30/2017, Bing Li
19    public class ChatPartnerRequestThread extends RequestQueue<ChatPartnerRequest,
20        ChatPartnerStream, ChatPartnerResponse>
21    {
22        public ChatPartnerRequestThread(int maxTaskSize)
23        {
24            super(maxTaskSize);
25        }
26
27        @Override
28        public void run()
29        {
30            ChatPartnerStream request;
31            ChatPartnerResponse response;
32            PeerAccount account;
33            PeerChatAccount chatAccount;
34            while (!this.isShutdown())
35            {
36                while (!this.isEmpty())
37                {
38                    request = this.getRequest();
39                    if (PeerRegistry.SYSTEM().isExisted(request.getMessage().getPartnerKey()))
40                    {
41                        // Get the account. 04/16/2017, Bing Li
42                        account = PeerRegistry.SYSTEM().get(request.getMessage().getPartnerKey());
43                        if (AccountRegistry.APPLICATION().isAccountExisted(request.getMessage()
```

```
44                          .getPartnerKey()))
45                      {
46                          chatAccount = AccountRegistry.APPLICATION().getAccount(request
47                              .getMessage().getPartnerKey());
48                          // Generate the response. 04/16/2017, Bing Li
49                          response = new ChatPartnerResponse(account.getPeerKey(), account.getPeerName(),
50                              chatAccount.getDescription(), chatAccount.getPreference(), account.getIP(),
51                              account.getPeerPort());
52                      }
53                      else
54                      {
55                          // Generate the response. 04/16/2017, Bing Li
56                          response = new ChatPartnerResponse(account.getPeerKey(), account.getPeerName(),
57                              UtilConfig.EMPTY_STRING, UtilConfig.EMPTY_STRING, account.getIP(),
58                              account.getPeerPort());
59                      }
60                  }
61                  else
62                  {
63                      // Generate the response. 04/16/2017, Bing Li
64                      response = new ChatPartnerResponse(UtilConfig.EMPTY_STRING,
65                          UtilConfig.EMPTY_STRING, UtilConfig.EMPTY_STRING,
66                          UtilConfig.EMPTY_STRING, UtilConfig.EMPTY_STRING, UtilConfig.ZERO);
67                  }
68                  try
69                  {
70                      // Respond to the client. 04/16/2017, Bing Li
71                      this.respond(request.getOutStream(), request.getLock(), response);
72                  }
73                  catch (IOException e)
74                  {
75                      e.printStackTrace();
76                  }
77                  this.disposeMessage(request, response);
78              }
79              try
80              {
81                  // Wait for some time when the queue is empty. During the period and before the thread is killed,
82                  // some new requests might be received. If so, the thread can keep working. 02/15/2016, Bing Li
83                  this.holdOn(ServerConfig.REQUEST_THREAD_WAIT_TIME);
84              }
85              catch (InterruptedException e)
86              {
87                  e.printStackTrace();
88              }
89          }
90      }
91  }
```

List 6.37 The code of ChatPartnerRequestThread.java

The request queue creator, ChatPartnerRequestThreadCreator, is shown in List 6.38.

```
1   package com.greatfree.chat.registry;
2
3   import com.greatfree.chat.message.peer.ChatPartnerRequest;
4   import com.greatfree.chat.message.peer.ChatPartnerResponse;
5   import com.greatfree.chat.message.peer.ChatPartnerStream;
6   import com.greatfree.concurrency.RequestThreadCreatable;
7
8   /*
9    * The creator generates one instance of ChatPartnerRequestThread. It is invoked by the thread management
10   * mechanism, RequestDispatcher. 04/30/2017, Bing Li
11   */
12
13  // Created: 04/30/2017, Bing Li
14  public class ChatPartnerRequestThreadCreator implements RequestThreadCreatable
15      <ChatPartnerRequest, ChatPartnerStream, ChatPartnerResponse, ChatPartnerRequestThread>
16  {
17      @Override
18      public ChatPartnerRequestThread createRequestThreadInstance(int taskSize)
19      {
20          return new ChatPartnerRequestThread(taskSize);
21      }
```

```
}
```

List 6.38 The code of ChatPartnerRequestThreadCreator.java

2.7 ND

The SD, CSServer<ChatManDispatcher>, employs one ND to receive management notifications from an administrator. Table 6.19 exhibits the pattern in brief.

Pattern: ND (Notification Dispatcher)		
Class	NotificationDispatcher<ShutdownChatServerNotification, ShutdownChattingRegistryServerThread, ShutdownChattingRegistryServerThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(ShutdownChatServerNotification notification)	
Enclosed Pattern	Notification	ShutdownChatServerNotification
	NQ (Notification Queue)	ShutdownChattingRegistryServerThread
	NQC (Notification Queue Creator)	ShutdownChattingRegistryServerThreadCreator
Employed API	ServerMessage	
	NotificationQueue<Notification extends ServerMessage>	
	NotificationThreadCreatable<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>>	

Table 6.19 The pattern of ND for dispatching administration notifications

The notification, ShutdownChatServerNotification, is shown in List 6.39.

```
package com.greatfree.chat.message;

import com.greatfree.message.ServerMessage;

/*
 * The notification aims to notify the chatting server to shutdown. 01/20/2016, Bing Li
 */

// Created: 04/17/2017, Bing Li
public class ShutdownChatServerNotification extends ServerMessage
{
    private static final long serialVersionUID = -7450693179089567 32L;

    public ShutdownChatServerNotification()
    {
        super(ChatMessageType.SHUTDOWN_CHAT_SERVER_NOTIFICATION);
    }
}
```

List 6.39 The code of ShutdownChatServerNotification.java

The notification queue, ShutdownChattingRegistryServerThread, is shown in List 6.40. It is also implemented using the pattern of DWC. Compared with DWC for request queues, there are a little bit updates. The primary change is that it is not necessary to respond to the administrator. Table 6.20 exhibits the pattern of DWC.

```
package com.greatfree.chat.registry;

import java.io.IOException;

import com.greatfree.chat.message.ShutdownChatServerNotification;
import com.greatfree.concurrency.NotificationQueue;
import com.greatfree.exceptions.RemoteReadException;
import com.greatfree.testing.data.ServerConfig;
import com.greatfree.util.ServerStatus;

/*
```

```
12      * The thread deals with shutting down notification from the administrator. 04/30/2017, Bing Li
13      */
14
15     // Created: 04/30/2017, Bing Li
16     public class ShutdownChattingRegistryServerThread extends NotificationQueue
17         <ShutdownChatServerNotification>
18     {
19         public ShutdownChattingRegistryServerThread(int taskSize)
20         {
21             super(taskSize);
23         }
24
25         @Override
26         public void run()
27         {
28             ShutdownChatServerNotification notification;
29             while (!this.isShutdown())
30             {
31                 while (!this.isEmpty())
32                 {
33                     try
34                     {
35                         notification = this.getNotification();
36                         ServerStatus.FREE().setShutdown();
37                         // Stop the registry server. 06/04/2017, Bing Li
38                         RegistryServer.PEER().stop();
39                         this.disposeMessage(notification);
40                     }
41                     catch (InterruptedException | IOException | ClassNotFoundException |
42                         RemoteReadException e)
43                     {
44                         e.printStackTrace();
45                     }
46                 }
47                 try
48                 {
49                     // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
50                     this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
51                 }
52                 catch (InterruptedException e)
53                 {
54                     e.printStackTrace();
55                 }
56             }
57         }
58     }
```

List 6.40 The code of ShutdownChattingRegistryServerThread.java

Pattern: DWC (Double While Concurrency)		
Class	ShutdownChattingRegistryServerThread	
Singleton	No	
Method	void dispose()	
	void enqueue(ShutdownChatServerNotification notification)	
	ShutdownChatServerNotification getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	ShutdownChatServerNotification
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 6.20 The pattern of DWC for processing the administration notification

The notification queue creator, ShutdownChattingRegistryServerThreadCreator, is shown in List 6.41.

```
1      package com.greatfree.chat.registry;
2
3      import com.greatfree.chat.message.ShutdownChatServerNotification;
4      import com.greatfree.concurrency.NotificationThreadCreatable;
5
```

```
// Created: 04/30/2017, Bing Li
public class ShutdownChattingRegistryServerThreadCreator implements NotificationThreadCreatable
    <ShutdownChatServerNotification, ShutdownChattingRegistryServerThread>
{
    @Override
    public ShutdownChattingRegistryServerThread createNotificationThreadInstance(int taskSize)
    {
        return new ShutdownChattingRegistryServerThread(taskSize);
    }
}
```

List 6.41 The code of ShutdownChattingRegistryServerThreadCreator.java

2.8 A Summary

Until now, the code of the registry server is discussed completely. I have to say that there is no anything new in the code in terms of techniques. If you have grasped the skills I introduce in previous chapters, you should program the above the code fast since each line of the code related to distributed systems follows the code level patterns, the idioms, including SP, MS, SD, RD, ND and DWC. In the later sections of the chapter, you will learn something new. However, the fundamental ideas are the same. That is, get the APIs and patterns to relieve the development burden as well as retain the developers' skills. As a generic programming environment for cloud or large-scale distributed systems, GreatFree has reached the goal so far.

In addition, although there are no new programming techniques discussed in the section, the concept of registry server is crucial for the peer-based chatting system. In conclusion, any distributed systems should have such a component. The chatting system we program in Chapter 5 has such a component as well while it is implicitly implemented in the chatting server. However, for a peer-based system, the registry server should be explicitly established.

Finally, the last issue I want to indicate is that it is not necessary for a registry server to be implemented on the distributed component of CSServer, which can respond to clients or peers passively only. In some specific cases, it is possible to implement a registry server with the model of Peer, which will be discussed soon, to push registered information to peers actively. But you need to learn more knowledge about the distributed computing environment over the Internet in addition to GreatFree programming techniques.

3. Programming the Server Side of the Chatting Peer

Programming with Peer is the primary topic of this chapter. A new distributed component, Peer, plays the lead in the chatting system. Since you decide to work on a new distributed model, P2P (Peer-to-Peer), rather than the one of C/S (Client/Server), no frameworks are available for you to CPR, as what you do in Chapter 3 and Chapter 4. You have to program from scratch. That is what you do in Chapter 5. If you know the knowledge of P2P, you can solve the problem easily even though the workload is a little bit high. Fortunately, once if the P2P framework is available, you can add more functions on it and implement your specific applications. I have to emphasize that the process to program scratch is not so horrible because it is still a procedure in which you can imitate or CPR since the interaction between one client and one server

is the most fundamental one on which each complicated system is built. That is why the contents in Chapter 3 and Chapter 4 are always useful to you when programming with GreatFree.

Parameter	Explanation
String peerName	The name of the peer
String sysRegistryIP	The IP address of the registry server on the system level
int sysRegistryPort	The port number of the registry server on the system level
int listenerCount	The listening thread count for the TCP port, which is identical to the one, listenerCount, in Table 5.5
int listenerThreadPoolSize	The listening thread is managed by a shared thread pool, which might be employed by more than one scenarios. Its size is set by the argument. It is identical to the one, listenerThreadPoolSize, in Table 5.5
long listenerThreadKeepAliveTime	The listening thread should be killed when it is believed to be idle long enough. The time is called keeping alive time. It is identical to the one, listenerThreadKeepAliveTime , in Table 5.5
Dispatcher dispatcher	The dispatcher that is responsible for dispatching chatting messages to corresponding threads such that those messages are processed concurrently. Its instance is set as an argument of the chatting server. It is identical to the one, dispatcher, in Table 5.5
int clientPoolSize	The size of TCP client pool on the peer
int readerClientSize	The size of Reader client pool on the peer
long syncEventerIdleCheckDelay	The delay to check whether TCP clients are busy or idle in the TCP client pool. Since those clients are used for sending synchronous notifications, the delay is named as syncEventerIdleCheckDelay
long syncEventerIdleCheckPeriod	The period to check whether TCP clients are busy or idle in the TCP client pool. Since those clients are used for sending synchronous notifications, the delay is named as syncEventerIdleCheckPeriod
long syncEventerMaxIdleTime	The maximum idle time for a TCP client in the TCP client pool. If the value is exceeded, it indicates that the TCP client is idle and can be collected
int asyncEventQueueSize	Since the peer encloses one instance of AsyncRemoteEventer, it needs to set the parameters that are needed by the instance of AsyncRemoteEventer. The parameter, asyncEventQueueSize, is identical to the one, eventQueueSize, in Table 5.27. It is the length of the built-in queue of Eventer that takes notifications sequentially to be sent asynchronously. It is necessary to emphasize that AsyncRemoteEventer is also a thread pool to manage instances of Eventer, which the thread derived from NotificationObjectQueue
int asyncEventerSize	The maximum size of Eventer instances that can be created in the instance of AsyncRemoteEventer. The parameter is identical to the one, eventerSize, in Table 5.27
long asyncEventingWaitTime	When no notifications are available, the instance of AsyncRemoteEventer needs to wait for some time before disposing itself to save resources. The time is called, eventingWaitTime. It is identical to the one, eventingWaitTime, in Table 5.27
long asyncEventerWaitTime	AsyncRemoteEventer is a pool to manage instances of Eventer, which is actually an instance of NotificationObjectQueue. The thread needs to wait for some time when its queue is empty. That time is called, eventerWaitTime. The parameter is identical to the one, eventerWaitTime, in Table 5.27
int asyncEventerWaitRound	When no notifications should be processed, to ensure whether it is time to dispose itself, it is necessary to wait for a couple of times before doing that. One time is identical to the round. It indicates that the instance of AsyncRemoteEventer has waited for a couple of periods, each of which is equal to eventingWaitTime. If the value is higher, it represents that it should wait for more rounds, and vice versa. The parameter is identical to the one, waitRound, in Table 5.27
long asyncEventIdleCheckDelay	The instance of AsyncRemoteEventer needs to check whether alive instances of Eventer is idle long enough. The value is the delay before the periodical checking is started. It is identical to the one, idleCheckDelay, in Table 5.27
long asyncEventIdleCheckPeriod	The length of the period to check whether alive instances of Eventer are idle long enough. It is employed by the instance of AsyncRemoteEventer. The parameter is identical to the one, idleCheckPeriod, in Table 5.27
int clientThreadPoolSize	The peer has a built-in thread pool that manages threads inside it. The size of the pool is called clientThreadPoolSize
long clientThreadKeepAliveTime	The keeping alive time for the built-in thread pool on the peer
int schedulerPoolSize	The size of the thread pool inside the scheduler, which is used to check periodically whether eventers in the AysncRemoteEventer is idle or not. If idle, they should be collected. It is identical to the one, schedulerPoolSize, in Table 5.27
long schedulerKeepAliveTime	The keeping alive time for thread pool inside the scheduler for the instance of AsyncRemoteEventer. It is identical to the one, schedulerKeepAliveTime, in Table 5.27

Table 6.21 The parameters of the constructor of Peer

3.1 The Parameters of Peer

Its APIs are exhibited in Table 6.1 and Table 6.2. According to the tables, those parameters of the constructor of Peer should be explained at first. Table 6.21 introduces the senses of all of the parameters. But I need to mention that most of the parameters have been discussed in Table 5.5 about the parameters of CSServer, Table 5.27 about the parameters of AsyncRemoteEventer. Since the peer is made up with a server and a client conceptually, it should contain parameters from both of them. Moreover, as a full-fledged distributed node, a peer includes SyncRemoteEventer and RemoteReader, which are not implemented in the chatting client of Chapter 5, to interact with other peers. So its parameters are an aggregation of all of the parameters from a mature server and a mature client.

3.2 SP: Starting the Peer

Let us open the code of StartChatPeer.java, which is shown in List 6.42. As an aggregation of a server and a client, the code contains the lines which we have seen both on the server side (List 5.1) and the client side (List 5.41). Based on both of the server code and the client code, you can understand the code of List 6.42 easily. I want to indicate one difference for the SP of peer. In the traditional client code of List 5.41, there are lines to initialize a reader (Line 45) and an eventer (Line 52) as well as a scheduler (Line 48 and Line 58). However, they disappear in the code of the peer. First of all, the reader and the eventer are still needed. However, they are initialized inside the instance of Peer. In addition to them, the scheduler is not necessary since the interaction between peers can be performed by sending messages directly rather than through the assistance of the chatting server. So the peer do not need to send polling requests periodically. That is why the scheduler is gone from the code of starting point code of a peer.

```
1     package com.greatfree.chat.peer;
2
3     import java.io.IOException;
4     import java.util.Scanner;
5
6     import com.greatfree.chat.ClientMenu;
7     import com.greatfree.chat.MenuOptions;
8     import com.greatfree.exceptions.RemoteReadException;
9     import com.greatfree.testing.data.ServerConfig;
10    import com.greatfree.util.ServerStatus;
11    import com.greatfree.util.TerminateSignal;
12
13    // Created: 04/30/2017, Bing Li
14    public class StartChatPeer
15    {
16        public static void main(String[] args)
17        {
18            // Initialize the option which represents a user's intents of operations. 09/21/2014, Bing Li
19            int option = MenuOptions.NO_OPTION;
20
21            // Initialize a command input console for users to interact with the system. 09/21/2014, Bing Li
22            Scanner in = new Scanner(System.in);
23            String optionStr;
25            System.out.println("Tell me your user name: ");
27            String username = in.nextLine();
28            System.out.println("Tell me your partner: ");
29            String partner = in.nextLine();
30            ChatMaintainer.PEER().init(username, partner);
31            System.out.println("Chatting peer starting up ...");
32
33            try
```

```
34          {
35              // Start up the peer. 06/04/2017, Bing Li
36              ChatPeerSingleton.PEER().start(username);
37          }
38          catch (IOException | ClassNotFoundException | RemoteReadException e)
39          {
40              e.printStackTrace();
41          }
42
43          System.out.println("Chatting peer started ...");
44
45          // Keep the loop running to interact with users until an end option is selected. 09/21/2014, Bing Li
46          while (option != MenuOptions.QUIT)
47          {
48              // Display the menu to users. 09/21/2014, Bing Li
49              ClientUI.PEER().printMenu();
50              // Input a string that represents users' intents. 09/21/2014, Bing Li
51              optionStr = in.nextLine();
52              try
53              {
54                  // Convert the input string to integer. 09/21/2014, Bing Li
55                  option = Integer.parseInt(optionStr);
56                  System.out.println("Your choice: " + option);
57                  // Send the option to the server. 09/21/2014, Bing Li
58                  ClientUI.PEER().send(option);
59              }
60              catch (NumberFormatException | ClassNotFoundException | RemoteReadException
61                   | IOException e)
62              {
63                  option = MenuOptions.NO_OPTION;
64                  System.out.println(ClientMenu.WRONG_OPTION);
65              }
66          }
67
68          ServerStatus.FREE().setShutdown();
69          try
70          {
71              // Shutdown the peer. 06/04/2017, Bing Li
72              ChatPeerSingleton.PEER().stop();
73          }
74          catch (IOException | InterruptedException | ClassNotFoundException | RemoteReadException e)
75          {
76              e.printStackTrace();
77          }
78
79          // After the server is started, the loop check whether the flag of terminating is set. If the terminating flag
80          // is true, the process is ended. Otherwise, the process keeps running. 08/22/2014, Bing Li
81          while (!TerminateSignal.SIGNAL().isTerminated())
82          {
83              try
84              {
85                  // If the terminating flag is false, it is required to sleep for some time. Otherwise, it might cause the
86                  // high CPU usage. 08/22/2014, Bing Li
87                  Thread.sleep(ServerConfig.TERMINATE_SLEEP);
88              }
89              catch (InterruptedException e)
90              {
91                  e.printStackTrace();
92              }
93          }
94          in.close();
95      }
96  }
```

List 6.42 The code of StartChatPeer.java

Pattern: SP (Starting Point)		
Class	StartChatPeer	
Singleton	Yes	
Method	void main(String[] args)	
Enclosed Pattern	MS (Main Server)	ChatPeerSingleton
Employed API	TerminalSignal	

Table 6.22 The pattern of SP for the chatting peer

Although the code is more complicated than a traditional server for the client code, the primary components of the SP for a server still exists. So we need to exhibit the pattern of SP for the peer in Table 6.22 as follows.

3.3 MS: The Chat Peer Singleton

As the pattern of MS, the code in List 6.43 has the same structure as the one in List 5.2. However, the API employed is different. Additionally, for the registry server in the peer-based chatting system, the tasks in the method of start() become complicated. Table 6.23 summarizes the pattern of MS fro the chatting peer.

Pattern: MS (Main Server)		
Class	ChatPeerSingleton	
Singleton	Yes	
Method	void start()	
	void stop()	
Enclosed Pattern	SD (Server Dispatcher)	Peer<ChatServerDispatcher>
		CSServer<ChatManServerDispatcher>
Employed API	Peer<Dispatcher extends ServerDispatcher>	
	CSServer<Dispatcher extends ServerDispatcher>	

Table 6.23 The pattern of MS for the chatting peer

```
1   package com.greatfree.chat.peer;
2
3   import java.io.IOException;
4
5   import com.greatfree.chat.ChatConfig;
6   import com.greatfree.chat.message.peer.AddPartnerNotification;
7   import com.greatfree.chat.message.peer.ChatNotification;
8   import com.greatfree.chat.message.peer.ChatPartnerRequest;
9   import com.greatfree.chat.message.peer.ChatPartnerResponse;
10  import com.greatfree.chat.message.peer.ChatRegistryRequest;
11  import com.greatfree.chat.message.peer.ChatRegistryResponse;
12  import com.greatfree.exceptions.RemoteReadException;
13  import com.greatfree.message.PortRequest;
14  import com.greatfree.message.PortResponse;
15  import com.greatfree.server.CSServer;
16  import com.greatfree.server.Peer;
17  import com.greatfree.testing.data.ServerConfig;
18  import com.greatfree.util.TerminateSignal;
19  import com.greatfree.util.UtilConfig;
20
21  /*
22   * This is the chatting peer. As an independent distributed node, each peer is able to interact with each other
23   * without a centralized polling server. That is, they can send messages instantly. 04/30/2017, Bing Li
24   */
25
26  // Created: 04/30/2017, Bing Li
27  public class ChatPeerSingleton
28  {
29      private Peer<ChatServerDispatcher> chatPeer;
30      private CSServer<ChatManServerDispatcher> manServer;
31
32      private ChatPeerSingleton()
33      {
34      }
35
36      private static ChatPeerSingleton instance = new ChatPeerSingleton();
37
38      public static ChatPeerSingleton PEER()
39      {
40          if (instance == null)
41          {
42              instance = new ChatPeerSingleton();
43              return instance;
44          }
```

```
45          else
46          {
47              return instance;
48          }
49      }
50
51      /*
52       * Stop the chatting peer. 05/12/2017, Bing Li
53       */
54      public void stop() throws IOException, InterruptedException, ClassNotFoundException,
55          RemoteReadException
56      {
57          // Set the terminating signal. 11/25/2014, Bing Li
58          TerminateSignal.SIGNAL().setTerminated();
59          // Stop the peer. 04/30/2017, Bing Li
60          this.chatPeer.stop();
61          // Stop the management server. 04/30/2017, Bing Li
62          this.manServer.stop();
63      }
64
65      /*
66       * Start the chatting peer. 05/12/2017, Bing Li
67       */
68      public void start(String username) throws IOException, ClassNotFoundException,
69          RemoteReadException
70      {
71          // Initialize the peer. 06/05/2017, Bing Li
72          this.chatPeer = new Peer.PeerBuilder<ChatServerDispatcher>()
73              .peerPort(ChatConfig.CHAT_SERVER_PORT)
74              .peerName(username)
75              .sysRegistryServerIP(ChatConfig.CHAT_REGISTRY_ADDRESS)
76              .sysRegistryServerPort(UtilConfig.PEER_REGISTRY_PORT)
77              .listenerCount(ServerConfig.LISTENING_THREAD_COUNT)
78              .listenerThreadPoolSize(ServerConfig.SHARED_THREAD_POOL_SIZE)
79              .listenerThreadKeepAliveTime
80                  (ServerConfig.SHARED_THREAD_POOL_KEEP_ALIVE_TIME)
81              .dispatcher(new ChatServerDispatcher(ChatConfig.DISPATCHER_THREAD_POOL_SIZE,
82                  ChatConfig.DISPATCHER_THREAD_POOL_KEEP_ALIVE_TIME,
83                  ChatConfig.SCHEDULER_THREAD_POOL_SIZE,
84                  ChatConfig.SCHEDULER_THREAD_POOL_KEEP_ALIVE_TIME))
85              .clientPoolSize(ChatConfig.CLIENT_POOL_SIZE)
86              .readerClientSize(ChatConfig.READER_CLIENT_SIZE)
87              .syncEventerIdleCheckDelay(ChatConfig.SYNC_EVENTER_IDLE_CHECK_DELAY)
88              .syncEventerIdleCheckPeriod(ChatConfig.SYNC_EVENTER_IDLE_CHECK_PERIOD)
89              .syncEventerMaxIdleTime(ChatConfig.SYNC_EVENTER_MAX_IDLE_TIME)
90              .asyncEventQueueSize(ChatConfig.ASYNC_EVENT_QUEUE_SIZE)
91              .asyncEventerSize(ChatConfig.ASYNC_EVENTER_SIZE)
92              .asyncEventingWaitTime(ChatConfig.ASYNC_EVENTING_WAIT_TIME)
93              .asyncEventerWaitTime(ChatConfig.ASYNC_EVENTER_WAIT_TIME)
94              .asyncEventerWaitRound(ChatConfig.ASYNC_EVENTER_WAIT_ROUND)
95              .asyncEventIdleCheckDelay(ChatConfig.ASYNC_EVENT_IDLE_CHECK_DELAY)
96              .asyncEventIdleCheckPeriod(ChatConfig.ASYNC_EVENT_IDLE_CHECK_PERIOD)
97              .clientThreadPoolSize(ChatConfig.CLIENT_THREAD_POOL_SIZE)
98              .clientThreadKeepAliveTime(ChatConfig.CLIENT_THREAD_KEEP_ALIVE_TIME)
99              .schedulerPoolSize(ChatConfig.SCHEDULER_THREAD_POOL_SIZE)
100             .schedulerKeepAliveTime(ChatConfig.SCHEDULER_THREAD_POOL_KEEP_ALIVE_TIME)
101             .build();
102
103         // Start the peer. 04/30/2017, Bing Li
104         this.chatPeer.start();
105
106         // Get the port for the management server. It is required when multiple peers run on
107         // the same node. 05/12/2017, Bing Li
108         PortResponse response = (PortResponse)this.chatPeer.read
109             (ChatConfig.CHAT_REGISTRY_ADDRESS, UtilConfig.PEER_REGISTRY_PORT,
110             new PortRequest(ChatMaintainer.PEER().getLocalUserKey(),
111             ChatConfig.PEER_ADMIN_PORT_KEY, this.chatPeer.getPeerIP(),
112             ChatConfig.CHAT_ADMIN_PORT));
113
114         // Initialize the chat management server. 04/30/2017, Bing Li
115         this.manServer = new CSServer.CSServerBuilder<ChatManServerDispatcher>()
116             .port(response.getPort())
117             .listenerCount(ServerConfig.SINGLE_THREAD_COUNT)
118             .listenerThreadPoolSize(ServerConfig.SHARED_THREAD_POOL_SIZE)
119             .listenerThreadKeepAliveTime
120                 (ServerConfig.SHARED_THREAD_POOL_KEEP_ALIVE_TIME)
121             .dispatcher(new ChatManServerDispatcher
```

```
122                 (ChatConfig.DISPATCHER_THREAD_POOL_SIZE,
123                 ChatConfig.DISPATCHER_THREAD_POOL_KEEP_ALIVE_TIME,
124                 ChatConfig.SCHEDULER_THREAD_POOL_SIZE,
125                 ChatConfig.SCHEDULER_THREAD_POOL_KEEP_ALIVE_TIME))
126             .build();
127
128         // Start the management server. 04/30/2017, Bing Li
129         this.manServer.start();
130     }
131
132     /*
133      * Register the chatting user. 06/05/2017, Bing Li
134      */
135     public ChatRegistryResponse registerChat(String localUserKey, String localUserName, String description,
136         String preference) throws ClassNotFoundException, RemoteReadException, IOException
137     {
138         return (ChatRegistryResponse)this.chatPeer.read(ChatConfig.CHAT_REGISTRY_ADDRESS,
139             ChatConfig.CHAT_REGISTRY_PORT, new ChatRegistryRequest(localUserKey,
140             localUserName, description, preference));
141     }
142
143     /*
144      * Search one potential chatting partner. 06/05/2017, Bing Li
145      */
146     public ChatPartnerResponse searchUser(String userKey) throws ClassNotFoundException,
147         RemoteReadException, IOException
148     {
149         return (ChatPartnerResponse)this.chatPeer.read(ChatConfig.CHAT_REGISTRY_ADDRESS,
150             ChatConfig.CHAT_REGISTRY_PORT, new ChatPartnerRequest(userKey));
151     }
152
153     /*
154      * Notify the potential partner that the local user wants to chat. 06/05/2017, Bing Li
155      */
156     public void notifyAddFriend(String ip, int port)
157     {
158         this.chatPeer.asyncNotify(ip, port, new AddPartnerNotification
159             (ChatMaintainer.PEER().getLocalUsername(), ChatMaintainer.PEER().getPartner(),
160             "Hello, I want to chat with you!"));
161     }
162
163     public void notifyChat(String ip, int port, String message)
164     {
165         this.chatPeer.asyncNotify(ip, port, new ChatNotification(message, ChatMaintainer.PEER()
166             .getLocalUsername()));
167     }
168 }
```

List 6.43 The code of ChatPeerSingleton.java

Line 29 declares the instance, chatPeer, of the new API, Peer, which we present at the beginning of the chapter. The instance accomplishes the primary chatting service for the local user.

Line 30 declares one instance of CSServer. It does not make contribution to the chatting. Similar to the one in Line 26 of List 6.2, the instance of CSServer here is responsible for receiving remote administration instructions to manage the local peer.

Line 32~49 defines the singleton for the chatting peer.

Line 54~63 terminates the chatting peer with the method, stop().

Line 68~130 initializes the peer. It involves multiple tasks to get the peer ready for chatting.

Line 72~101 initializes the instance of Peer, which is the single one for the chatting process for one participant. It has a bunch of parameters to be initialized, as shown in Table 6.21. In addition, since ServerDispatcher is one component for Peer, it is necessary to initialize here together in Line 81~84. For those parameters, you can refer to Table 5.6. Having more parameters is one important difference of Peer from CSServer.

Line 104 starts the instance of Peer.

Line 108~112 sends a request to the registry server to ensure whether the port, ChatConfig.CHAT_ADMIN_PORT, is idle such that it can be employed for the instance of the CSServer that waits for remote notifications from the administrator. If the local computing node has only one chatting peer, the line is not necessary. However, since it is possible that multiple peers run concurrently on the same node, the IP address must get conflicts without the coordination of the registry server. Compared with the chatting client in Section 3 of Chapter 5, the node does not need to obtain such a port. It proves that Peer is a server other than a traditional client. This is also distinguished.

After an idle port is obtained from the registry server, Line 115~126 initializes the instance of CSServer for the administration of the local node.

Line 129 starts the instance of CSServer for administration.

Line 135~167 presents the primary different from a traditional server since the lines contain four methods to send requests to the register server and send notifications to its chatting partner. Both of the two types of interactions are active rather than passive. All of them are based on the methods of Peer in Table 6.2.

Line 135~141 defines the method to register the local user for the chatting. The request, ChatRegistryRequest, is sent to the registry server and one response, ChatRegistryResponse, is obtained.

Line 146~151 defines the method to search one potential chatting partner from the registry server.

After the potential chatting partner is obtained, Line 156~161 sends a notification to the partner directly to let him know whether a potential chatting can be performed.

Line 163~167 defines the method to send one chatting message from the local peer to the remote one, i.e., the chatting partner.

According to the above descriptions, you can learn that programming with Peer is straightforward. In essence, one partial of the code (Line 135~167) is identical to the code of ChatReader.java in List 5.42 and the one of ChatEventer.java in List 5.43 for the chatting client. On the other hand, the other portion of code (Line 29~130) looks like a chatting server in List 5.2. It is easy to explain the phenomenon. That is, the Peer is an aggregation that includes the APIs of the traditional client as well as the traditional server. However, the difference is also apparent since the registry server,

which is established explicitly rather than implicitly, should be interacted when the chatting peer is started up.

Incidentally, do you have any questions so far? Do you remember that there is a registry, PeerRegistry (List 6.3), on the registry server? If so, take a check at Line 24 in List 6.2. Following the code over there, you can find List 6.6. As we have discussed, the peer registry is critical to the peer-based distributed system. But why are there no relevant requests, such as RegisterPeerRequest (List 6.10) to the register server in the above code? Is the chatting peer really ready for the coming chatting?

That is a good question. I need to indicate that the task is performed inside the peer when it is started at Line 104 in List 6.43. Since each peer needs to accomplish the registry before performing any applications on the P2P infrastructure, the mandatory job is done by the system and hidden from developers.

3.4 SD: The Server Side of the Peer

A server is always more complicated than a client in terms of the workload of coding at least. Now let us check the code in the pattern of SD, which is the typical idiom when programming a server.

3.4.1 SD for Chatting

List 6.44 shows the code of ChatServerDispatcher.java for the SD.

You can take a look at the code. It is no doubt that the structure of the code is exactly the same as the ones of List 6.6, List 6.7 and List 6.8. All of them are programmed in the pattern of SD. The same thing happens in List 5.9 and List 5.10. According to those practices, we can summarize again, the pattern of SD is always the stable structure for server side programming. Table 6.24 presents the pattern of SD for the chatting peer.

```
1    package com.greatfree.chat.peer;
2
3    import java.util.Calendar;
4
5    import com.greatfree.chat.message.ChatMessageType;
6    import com.greatfree.chat.message.peer.AddPartnerNotification;
7    import com.greatfree.chat.message.peer.ChatNotification;
8    import com.greatfree.concurrency.NotificationDispatcher;
9    import com.greatfree.message.ServerMessage;
10   import com.greatfree.remote.OutMessageStream;
11   import com.greatfree.server.ServerDispatcher;
12   import com.greatfree.testing.data.ServerConfig;
13
14   /*
15    * The server dispatcher is installed on the Peer rather than a CSServer. Therefore, the messages to be
16    * handled become different. 04/30/2017, Bing Li
17    */
18
19   // Created: 04/30/2017, Bing Li
20   public class ChatServerDispatcher extends ServerDispatcher
21   {
22       // Declare a notification dispatcher to add partner invitation to the peer when such
23       // a notification is received. 04/18/2016, Bing Li
24       private NotificationDispatcher<AddPartnerNotification, AddPartnerThread, AddPartnerThreadCreator>
25           partnerNotificationDispatcher;
```

```
26          // Declare a notification dispatcher to display chats to the peer when such a notification is
27          // received. 04/18/2016, Bing Li
28          private NotificationDispatcher<ChatNotification, ChatThread, ChatThreadCreator>
29              chatNotificationDispatcher;
30
31          public ChatServerDispatcher(int threadPoolSize, long threadKeepAliveTime, int schedulerPoolSize,
32              long schedulerKeepAliveTime)
33          {
34              super(threadPoolSize, threadKeepAliveTime, schedulerPoolSize, schedulerKeepAliveTime);
35
36              this.partnerNotificationDispatcher = new NotificationDispatcher.NotificationDispatcherBuilder
37                  <AddPartnerNotification, AddPartnerThread, AddPartnerThreadCreator>()
38                  .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
39                  .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
40                  .threadCreator(new AddPartnerThreadCreator())
41                  .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
42                  .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
43                  .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
44                  .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
45                  .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
46                  .scheduler(super.getSchedulerPool())
47                  .build();
48
49              this.chatNotificationDispatcher = new NotificationDispatcher.NotificationDispatcherBuilder
50                  <ChatNotification, ChatThread, ChatThreadCreator>()
51                  .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
52                  .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
53                  .threadCreator(new ChatThreadCreator())
54                  .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
55                  .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
56                  .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
57                  .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
58                  .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
59                  .scheduler(super.getSchedulerPool())
60                  .build();
61          }
62
63          /*
64           * Shut down the server message dispatcher. 04/15/2017, Bing Li
65           */
66          public void shutdown() throws InterruptedException
67          {
68              this.partnerNotificationDispatcher.dispose();
69              this.chatNotificationDispatcher.dispose();
70              super.shutdown();
71          }
72
73          /*
74           * Process the available messages in a concurrent way. 04/17/2017, Bing Li
75           */
76          public void consume(OutMessageStream<ServerMessage> message)
77          {
78              // Check the types of received messages. 04/17/2017, Bing Li
79              switch (message.getMessage().getType())
80              {
81                  case ChatMessageType.PEER_ADD_PARTNER_NOTIFICATION:
82                      System.out.println("PEER_ADD_PARTNER_NOTIFICATION received @" +
83                          Calendar.getInstance().getTime());
84                      // Check whether the adding chatting partner notification dispatcher is ready
85                      // or not. 02/15/2016, Bing Li
86                      if (!this.partnerNotificationDispatcher.isReady())
87                      {
88                          // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
89                          super.execute(this.partnerNotificationDispatcher);
90                      }
91                      // Enqueue the instance of AddPartnerNotification into the dispatcher for
92                      // concurrent processing. 02/15/2016, Bing Li
93                      this.partnerNotificationDispatcher.enqueue((AddPartnerNotification)message.getMessage());
94                      break;
95
96                  case ChatMessageType.PEER_CHAT_NOTIFICATION:
97                      System.out.println("PEER_CHAT_NOTIFICATION received @" +
98                          Calendar.getInstance().getTime());
99                      // Check whether the chatting notification dispatcher is ready or not. 02/15/2016, Bing Li
100                     if (!this.chatNotificationDispatcher.isReady())
101                     {
102                         // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
```

```
103                     super.execute(this.chatNotificationDispatcher);
104                 }
105                 // Enqueue the instance of ChatNotification into the dispatcher for concurrent
106                 // processing. 02/15/2016, Bing Li
107                 this.chatNotificationDispatcher.enqueue((ChatNotification)message.getMessage());
108                 break;
109             }
110         }
111     }
```

List 6.44 The code of ChatServerDispatcher.java

Pattern: SD (Server Dispatcher)		
Class	Peer<ChatServerDispatcher>	
Singleton	No	
Method	void consume(OutMessageStream<ServerMessage> message)	
	void shutdown()	
Enclosed Pattern	ND (Notification Dispatcher)	NotificationDispatcher<AddPartnerNotification, AddPartnerThread, AddPartnerThreadCreator>
		NotificationDispatcher<ChatNotification, ChatThread, ChatThreadCreator>
Employed API	ServerDispatcher	
	NotificationDispatcher<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>, ThreadCreator extends NotificationThreadCreatable<Notification, NotificationThread>>	

Table 6.24 The pattern of SD for the chatting peer

Line 24 declares one instance of ND to process the notifications, AddPartnerNotification, of adding chatting partners.

Line 28 declares one instance of ND to process the notifications, ChatNotification, of new received chatting messages.

Line 31~61 is the constructor of the SD. I believe you have totally grasped the structure. If still not, that is fine. Just review Section 2.3 of Chapter 5. The pattern of SD is so important and so frequently used.

Line 66 defines the method, shutdown() to shut down the dispatcher.

Line 76~110 define the method, consume(), which dispatches incoming messages to specific NDs in order to process them concurrently.

Since we have seen SD so many times, we do not need to explain more.

3.4.2 SD for Management

Another SD for the chatting peer is not so related to the chatting service. It has counterparts in the previous example, the chatting system on the C/S model. In Section 2.4 of Chapter 5, the code of List 5.10 also works for management rather than the chatting. But I indicate again that it is important in practice since we need to manage the computing system remotely through this way. When programming an application in the real world, you need to strengthen it.

List 6.45 shows the code of ChatManServerDispatcher.java for the peer. Table 6.25 presents the pattern of SD for the chatting management. The ND receives one notification, ShutdownChatServerNotification, from a remote administrator to shut

down the chatting peer. In practice, you can add more remote commands to the SD to manage the peer-based application.

```
1   package com.greatfree.chat.peer;
2
3   import java.util.Calendar;
4
5   import com.greatfree.chat.message.ChatMessageType;
6   import com.greatfree.chat.message.ShutdownChatServerNotification;
7   import com.greatfree.concurrency.NotificationDispatcher;
8   import com.greatfree.message.ServerMessage;
9   import com.greatfree.remote.OutMessageStream;
10  import com.greatfree.server.ServerDispatcher;
11  import com.greatfree.testing.data.ServerConfig;
12
13  /*
14   * The server dispatcher extends com.greatfree.server.ServerDispatcher to implement a sub server
15   * dispatcher for administration. 04/15/2017, Bing Li
16   */
17
18  // Created: 05/01/2017, Bing Li
19  public class ChatManServerDispatcher extends ServerDispatcher
20  {
21      // Declare a notification dispatcher to shutdown the server when such a notification
22      // is received. 04/18/2016, Bing Li
23      private NotificationDispatcher<ShutdownChatServerNotification, ShutdownChattingPeerThread,
24          ShutdownChattingPeerThreadCreator> shutdownNotificationDispatcher;
25
26      public ChatManServerDispatcher(int threadPoolSize, long threadKeepAliveTime, int schedulerPoolSize,
27          long schedulerKeepAliveTime)
28      {
29          super(threadPoolSize, threadKeepAliveTime, schedulerPoolSize, schedulerKeepAliveTime);
30
31          this.shutdownNotificationDispatcher = new NotificationDispatcher.NotificationDispatcherBuilder
32              <ShutdownChatServerNotification, ShutdownChattingPeerThread,
33              ShutdownChattingPeerThreadCreator>()
34              .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
35              .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
36              .threadCreator(new ShutdownChattingPeerThreadCreator())
37              .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
38              .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
39              .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
40              .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
41              .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
42              .scheduler(super.getSchedulerPool())
43              .build();
44      }
45
46      /*
47       * Shut down the server message dispatcher. 04/15/2017, Bing Li
48       */
49      public void shutdown() throws InterruptedException
50      {
51          this.shutdownNotificationDispatcher.dispose();
52          super.shutdown();
53      }
54
55      /*
56       * Process the available messages in a concurrent way. 04/17/2017, Bing Li
57       */
58      public void consume(OutMessageStream<ServerMessage> message)
59      {
60          // Check the types of received messages. 04/17/2017, Bing Li
61          switch (message.getMessage().getType())
62          {
63              case ChatMessageType.SHUTDOWN_CHAT_SERVER_NOTIFICATION:
64                  System.out.println("SHUTDOWN_CHAT_SERVER_NOTIFICATION received @" +
65                      Calendar.getInstance().getTime());
66                  // Check whether the shutdown notification dispatcher is ready or not. 02/15/2016, Bing Li
67                  if (!this.shutdownNotificationDispatcher.isReady())
68                  {
69                      // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
70                      super.execute(this.shutdownNotificationDispatcher);
71                  }
72                  // Enqueue the instance of ShutdownChatServerNotification into the dispatcher for
```

```
73                  // concurrent processing. 02/15/2016, Bing Li
74                  this.shutdownNotificationDispatcher.enqueue((ShutdownChatServerNotification)
75                      message.getMessage());
76                  break;
77              }
78          }
79      }
```

List 6.45 The code of ChatManServerDispatcher.java

Pattern: SD (Server Dispatcher)		
Class	CSServer<ChatManServerDispatcher>	
Singleton	No	
Method	void consume(OutMessageStream<ServerMessage> message)	
	void shutdown()	
Enclosed Pattern	ND (Notification Dispatcher)	NotificationDispatcher<ShutdownChatServerNotification, ShutdownChattingPeerThread, ShutdownChattingPeerThreadCreator>
Employed API	ServerDispatcher	
	NotificationDispatcher<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>, ThreadCreator extends NotificationThreadCreatable<Notification, NotificationThread>>	

Table 6.25 The pattern of ND for the chatting management

3.5 ND

For the specific application of chatting, the server side has only one type of patterns, i.e., ND. We just present the code and gives a simple explanation to them since we have done that a lot in the previous chapters. You can refer to the detailed specifications for them in Section 2.6 of Chapter 5. We also encounter the pattern of ND in Section 2.7 of this chapter.

3.5.1 The ND for Adding Partner

The first ND, partnerNotificationDispatcher, is declared in Line 24 of List 6.44. Its pattern is illustrated in Table 6.26.

Pattern: ND (Notification Dispatcher)		
Class	NotificationDispatcher<AddPartnerNotification, AddPartnerThread, AddPartnerThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(AddPartnerNotification notification)	
Enclosed Pattern	Notification	AddPartnerNotification
	NQ (Notification Queue)	AddPartnerThread
	NQC (Notification Queue Creator)	AddPartnerThreadCreator
Employed API	ServerMessage	
	NotificationQueue<Notification extends ServerMessage>	
	NotificationThreadCreatable<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>>	

Table 6.26 The pattern of ND for adding chatting partners

The notification of the ND is called, AddPartnerNotification, which is shown in List 6.46. The notification contains the sender user name (localUserName), the chatting partner name (partnerName) and the invitation message (invitation).

```
1    package com.greatfree.chat.message.peer;
2
```

```
3     import com.greatfree.chat.message.ChatMessageType;
4     import com.greatfree.message.ServerMessage;
5
6     /*
7      * The message encloses the chatting partner to be added. 04/15/2017, Bing Li
8      */
9
10    // Created: 05/02/2017, Bing Li
11    public class AddPartnerNotification extends ServerMessage
12    {
13        private static final long serialVersionUID = -4853146777469532800L;
14
15        private String localUserName;
16        private String partnerName;
17        private String invitation;
18
19        public AddPartnerNotification(String localUserName, String partnerName, String invitation)
20        {
21            super(ChatMessageType.PEER_ADD_PARTNER_NOTIFICATION);
22            this.localUserName = localUserName;
23            this.partnerName = partnerName;
24            this.invitation = invitation;
25        }
26
27        public String getLocalUserName()
28        {
29            return this.localUserName;
30        }
31
32        public String getPartnerName()
33        {
34            return this.partnerName;
35        }
36
37        public String getInvitation()
38        {
39            return this.invitation;
40        }
41    }
```

List 6.46 The code of AddPartnerNotification.java

The notification queue for the thread is called, AddPartnerThread, which is shown in List 6.47. Repeat again, the pattern DWC is utilized in the thread. Table 6.27 illustrates the pattern.

```
1     package com.greatfree.chat.peer;
2
3     import com.greatfree.chat.message.peer.AddPartnerNotification;
4     import com.greatfree.concurrency.NotificationQueue;
5     import com.greatfree.testing.data.ServerConfig;
6
7     /*
8      * The thread deals with adding chatting friend notification from the other peer. 04/30/2017, Bing Li
9      */
10
11    // Created: 05/02/2017, Bing Li
12    public class AddPartnerThread extends NotificationQueue<AddPartnerNotification>
13    {
14        public AddPartnerThread(int taskSize)
15        {
16            super(taskSize);
17        }
18
19        @Override
20        public void run()
21        {
22            AddPartnerNotification notification;
23            while (!this.isShutdown())
24            {
25                while (!this.isEmpty())
26                {
27                    try
```

```
28                  {
29                      notification = this.getNotification();
30                      // Just show the invitation on the screen. 06/11/2017, Bing Li
31                      System.out.println(notification.getLocalUserName() + " says: " + notification.getInvitation());
32                      this.disposeMessage(notification);
33                  }
34                  catch (InterruptedException e)
35                  {
36                      e.printStackTrace();
37                  }
38              }
39              try
40              {
41                  // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
42                  this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
43              }
44              catch (InterruptedException e)
45              {
46                  e.printStackTrace();
47              }
48          }
49      }
50  }
```

List 6.47 The code of AddPartnerThread.java

Pattern: DWC (Double While Concurrency)		
Class	AddPartnerThread	
Singleton	No	
Method	void dispose()	
	void enqueue(AddPartnerNotification notification)	
	AddPartnerNotification getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	AddPartnerNotification
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 6.27 The pattern of DWC for processing the notifications of adding partner concurrently

The notification queue creator is named as AddPartnerThreadCreator, which is shown in List 6.48.

```
1   package com.greatfree.chat.peer;
2
3   import com.greatfree.chat.message.peer.AddPartnerNotification;
4   import com.greatfree.concurrency.NotificationThreadCreatable;
5
6   // Created: 05/02/2017, Bing Li
7   public class AddPartnerThreadCreator implements NotificationThreadCreatable<AddPartnerNotification,
8       AddPartnerThread>
9   {
10      @Override
11      public AddPartnerThread createNotificationThreadInstance(int taskSize)
12      {
13          return new AddPartnerThread(taskSize);
14      }
15  }
```

List 6.48 The code of AddPartnerThreadCreator.java

3.5.2 The ND for Chatting Notifications

The ND, chatNotificationDispatcher, defined in Line 28 of List 6.44, for chatting notifications is responsible for processing received chatting messages concurrently. The primary pattern, DWC, is illustrated in Table 6.28.

The notification of the ND is called, ChatNotification, which is shown in List 6.49. It contains the chatting message, message, and the message sender, senderName.

```
1   package com.greatfree.chat.message.peer;
2
3   import com.greatfree.chat.message.ChatMessageType;
4   import com.greatfree.message.ServerMessage;
5   import com.greatfree.util.UtilConfig;
6
7   // Created: 05/02/2017, Bing Li
8   public class ChatNotification extends ServerMessage
9   {
10      private static final long serialVersionUID = -4180547758940870394L;
11
12      // The chatting message. 04/27/2017, Bing Li
13      private String message;
14      // The sender name. 04/27/2017, Bing Li
15      private String senderName;
16
17      public ChatNotification(String message, String senderName)
18      {
19          super(ChatMessageType.PEER_CHAT_NOTIFICATION);
20          this.message = message;
21          this.senderName = senderName;
22      }
23
24      public ChatNotification(String message)
25      {
26          super(ChatMessageType.PEER_CHAT_NOTIFICATION);
27          this.message = message;
28          this.senderName = UtilConfig.EMPTY_STRING;
29      }
30
31      public String getMessage()
32      {
33          return this.message;
34      }
35
36      public String getSenderName()
37      {
38          return this.senderName;
39      }
40  }
```

List 6.49 The code of ChatNotification.java

Pattern: DWC (Double While Concurrency)		
Class	ChatThread	
Singleton	No	
Method	void dispose()	
	void enqueue(ChatNotification notification)	
	ChatNotification getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	ChatNotification
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 6.28 The pattern of DWC for processing chatting notification concurrently

The notification queue of the ND is named as ChatThread, which is illustrated in List 6.50. The DWC pattern is presented in Table 6.28.

```
1    package com.greatfree.chat.peer;
2
3    import com.greatfree.chat.message.peer.ChatNotification;
4    import com.greatfree.concurrency.NotificationQueue;
5    import com.greatfree.testing.data.ServerConfig;
6
7    /*
8    * The thread deals with the chatting notification from the other peer. 04/30/2017, Bing Li
9     */
10
11   // Created: 05/02/2017, Bing Li
12   public class ChatThread extends NotificationQueue<ChatNotification>
13   {
14       public ChatThread(int taskSize)
15       {
16           super(taskSize);
17       }
18
19       @Override
20       public void run()
21       {
22           ChatNotification notification;
23           while (!this.isShutdown())
24           {
25               while (!this.isEmpty())
26               {
27                   try
28                   {
29                       notification = this.getNotification();
30                       // Show the chatting message on the screen. 06/11/2017, Bing Li
31                       System.out.println(notification.getSenderName() + " says, " + notification.getMessage());
32                       this.disposeMessage(notification);
33                   }
34                   catch (InterruptedException e)
35                   {
36                       e.printStackTrace();
37                   }
38               }
39               try
40               {
41                   // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
42                   this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
43               }
44               catch (InterruptedException e)
45               {
46                   e.printStackTrace();
47               }
48           }
49       }
50   }
```

List 6.50 The code of ChatThread.java

The creator, ChatThreadCreator, of the notification queue is shown in List 6.51.

```
1    package com.greatfree.chat.peer;
2
3    import com.greatfree.chat.message.peer.ChatNotification;
4    import com.greatfree.concurrency.NotificationThreadCreatable;
5
6    // Created: 05/02/2017, Bing Li
7    public class ChatThreadCreator implements NotificationThreadCreatable<ChatNotification,  ChatThread>
8    {
9        @Override
10       public ChatThread createNotificationThreadInstance(int taskSize)
11       {
12           return new ChatThread(taskSize);
13       }
```

```
14    }
```

List 6.51 The code of ChatThreadCreator.java

3.5.3 The ND for Shutting Down the Chatting Peer

The last ND employed by the chatting system is called shutdownNotificationDispatcher, which is declared in Line 23 of List 6.45. The ND includes three components as others, including the notification of ShutdownChatServerNotification, the notification queue of ShutdownChatPeerThread and the notification queue creator of ShutdownChattingPeerThreadCreator.

The notification, ShutdownChatServerNotification, is shared with the chatting server. The code is shown in List 5.38 of Section 2.6.3. So here, I do not present it again.

Although the notification is identical to the ones for other servers, the thread should be different because it needs to stop different servers after receiving the notification. For that, the code, ShutdownChatPeerThread.java, is shown in List 6.52. For thread, it invokes the method of stop() of ChatPeerSingleton at Line 37. The pattern of DWC is summarized in Table 6.29.

```
1     package com.greatfree.chat.peer;
2
3     import java.io.IOException;
4
5     import com.greatfree.chat.message.ShutdownChatServerNotification;
6     import com.greatfree.concurrency.NotificationQueue;
7     import com.greatfree.exceptions.RemoteReadException;
8     import com.greatfree.testing.data.ServerConfig;
9     import com.greatfree.util.ServerStatus;
10
11    /*
12     * The thread deals with shutting down notification from the administrator. 04/23/2017, Bing Li
13     */
14
15    // Created: 05/01/2017, Bing Li
16    public class ShutdownChattingPeerThread extends NotificationQueue<ShutdownChatServerNotification>
17    {
18        public ShutdownChattingPeerThread(int taskSize)
19        {
20            super(taskSize);
21        }
22
23        @Override
24        public void run()
25        {
26            ShutdownChatServerNotification notification;
27            while (!this.isShutdown())
28            {
29                while (!this.isEmpty())
30                {
31                    try
32                    {
33                        notification = this.getNotification();
34                        // Set the server is shut down. 05/01/2017, Bing Li
35                        ServerStatus.FREE().setShutdown();
36                        // Shut down the chatting peer. 05/01/2017, Bing Li
37                        ChatPeerSingleton.PEER().stop();
38                        this.disposeMessage(notification);
39                    }
40                    catch (InterruptedException | IOException | ClassNotFoundException |
41                        RemoteReadException e)
42                    {
43                        e.printStackTrace();
44                    }
```

```
45                }
46                try
47                {
48                    // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
49                    this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
50                }
51                catch (InterruptedException e)
52                {
53                    e.printStackTrace();
54                }
55            }
56        }
57    }
```

List 6.52 The code of ShutdownChattingPeerThread.java

Pattern: DWC (Double While Concurrency)		
Class	ShutdownChattingPeerThread	
Singleton	No	
Method	void dispose()	
	void enqueue(ServerMessage notification)	
	ShutdownChatServerNotification getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	ShutdownChatServerNotification
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 6.29 The pattern of DWC for processing the notification of shutting the chatting peer

The creator of the notification queue is shown in List 6.53.

```
1     package com.greatfree.chat.peer;
2
3     import com.greatfree.chat.message.ShutdownChatServerNotification;
4     import com.greatfree.concurrency.NotificationThreadCreatable;
5
6     // Created: 05/01/2017, Bing Li
7     public class ShutdownChattingPeerThreadCreator implements NotificationThreadCreatable
8         <ShutdownChatServerNotification, ShutdownChattingPeerThread>
9     {
10        @Override
11        public ShutdownChattingPeerThread createNotificationThreadInstance(int taskSize)
12        {
13            return new ShutdownChattingPeerThread(taskSize);
14        }
15    }
```

List 6.53 The code of ShutdownChattingPeerThreadCreator.java

4. Programming the Client Side of the Chatting Peer

A peer is made up with two components, the server side and the client side. Section 3 introduces all of the code and patterns of the server of the chatting peer. You must wonder what the client side looks like. Before the code is presented to you, you can imagine for some time according to your past experiences. Fortunately, the client code follows the patterns of RR and RE as well. If you forget, you can review Section 3.2 and Section 3.3 of Chapter 5. However, since the API of Peer hides all of the details to invoke the eventing and requesting APIs, so you cannot see them explicitly. For that, the code of client side is almost related to upper level applications rather than any distributed APIs and patterns.

In fact, you have learned primary client codes at Line 135~167 of the code in List 6.43. The methods within the lines are called by the application code of the client, i.e., ClientUI. For that, the client programming for the peer becomes extremely straightforward.

4.1 Chatting Maintainer

The first application code we see is the one at Line 30 in the code of StartChatPeer.java (List 6.42). So let us take a look at the code of ChatMaintainer.java, which is shown in List 6.54. Since the chatting application is simple, the code just keeps the partner's information, such as the partner's name and key. The most important information is the IP and the port of the chatting partner, which is obtained from the registry server. Anyway, the code is the one you can program according to your requirements of the application. I do not need to spend much time on it.

```
1    package com.greatfree.chat.peer;
2
3    import com.greatfree.chat.server.ChatTools;
4    import com.greatfree.util.UtilConfig;
5
6    /*
7     * The class maintains the partner and the local user for the chatting. 04/24/2017, Bing Li
8     */
9
10   // Created: 05/02/2017, Bing Li
11   public class ChatMaintainer
12   {
13       private String localUserKey;
14       private String localUsername;
15       private String partner;
16       private String partnerKey;
17       private String partnerIP;
18       private int partnerPort;
19
20       private ChatMaintainer()
21       {
22           this.partnerIP = null;
23           this.partnerPort = UtilConfig.ZERO;
24       }
25
26       /*
27        * A singleton definition. 04/17/2017, Bing Li
28        */
29       private static ChatMaintainer instance = new ChatMaintainer();
30
31       public static ChatMaintainer PEER()
32       {
33           if (instance == null)
34           {
35               instance = new ChatMaintainer();
36               return instance;
37           }
38           else
39           {
40               return instance;
41           }
42       }
43
44       /*
45        * Dispose the free client pool. 04/17/2017, Bing Li
46        */
47       public void dispose()
48       {
49       }
50
51       public void init(String username, String partner)
```

```
52          {
53              this.localUsername = username;
54              this.localUserKey = ChatTools.getUserKey(username);
55              this.partner = partner;
56              this.partnerKey = ChatTools.getUserKey(partner);
57          }
58
59          public String getLocalUserKey()
60          {
61              return this.localUserKey;
62          }
63
64          public String getLocalUsername()
65          {
66              return this.localUsername;
67          }
68
69          public String getPartner()
70          {
71              return this.partner;
72          }
73
74          public String getPartnerKey()
75          {
76              return this.partnerKey;
77          }
78
79          public void setPartnerIP(String ip)
80          {
81              this.partnerIP = ip;
82          }
83
84          public String getPartnerIP()
85          {
86              return this.partnerIP;
87          }
88
89          public void setPartnerPort(int port)
90          {
91              this.partnerPort = port;
92          }
93
94          public int getPartnerPort()
95          {
96              return this.partnerPort;
97          }
98      }
```

List 6.54 The code of ChatMaintainer.java

4.2 Chatting UI

The next application code we encounter is the one, ClientUI, at Line 49 of StartChatPeer in List 6.42. It presents a simple UI for users to manipulate the chatting application. The code is illustrated in List 6.55.

```
1       package com.greatfree.chat.peer;
2
3       import java.io.IOException;
4       import java.util.Scanner;
5
6       import com.greatfree.chat.ChatOptions;
7       import com.greatfree.chat.ClientMenu;
8       import com.greatfree.chat.MenuOptions;
9       import com.greatfree.chat.message.peer.ChatPartnerResponse;
10      import com.greatfree.chat.message.peer.ChatRegistryResponse;
11      import com.greatfree.exceptions.RemoteReadException;
12
13      /*
14       * The class aims to print a menu list on the screen for users to interact with the client and
15       * communicate with the chatting peer. The menu is unique in the client such that it is implemented
```

```
16      * in the pattern of a singleton. 05/02/2017, Bing Li
17      */
18
19     // Created: 05/02/2017, Bing Li
20     public class ClientUI
21     {
22         private Scanner in = new Scanner(System.in);
23
24         /*
25          * Initialize. 04/23/2017, Bing Li
26          */
27         private ClientUI()
28         {
29         }
30
31         /*
32          * Initialize a singleton. 04/23/2017, Bing Li
33          */
34         private static ClientUI instance = new ClientUI();
35
36         public static ClientUI PEER()
37         {
38             if (instance == null)
39             {
40                 instance = new ClientUI();
41                 return instance;
42             }
43             else
44             {
45                 return instance;
46             }
47         }
48
49         public void dispose()
50         {
51             this.in.close();
52         }
53
54         /*
55          * Print the menu list on the screen. 04/23/2017, Bing Li
56          */
57         public void printMenu()
58         {
59             System.out.println(ClientMenu.MENU_HEAD);
60             System.out.println(ClientMenu.REGISTER_CHATTING + ChatMaintainer.PEER()
61                 .getLocalUsername());
62             System.out.println(ClientMenu.SEARCH_USER + ChatMaintainer.PEER().getPartner());
63             System.out.println(ClientMenu.ADD_FRIEND + ChatMaintainer.PEER().getPartner());
64             System.out.println(ClientMenu.START_CHATTING + ChatMaintainer.PEER().getPartner());
65             System.out.println(ClientMenu.QUIT);
66             System.out.println(ClientMenu.MENU_TAIL);
67             System.out.println(ClientMenu.INPUT_PROMPT);
68         }
69
70         /*
71          * Send the users' option to the chatting server. 04/23/2017, Bing Li
72          */
73         public void send(int option) throws ClassNotFoundException, RemoteReadException, IOException
74         {
75             ChatRegistryResponse chatRegistryResponse;
76             ChatPartnerResponse chatPartnerResponse = null;
77
78             switch (option)
79             {
80                 // Register the local user for chatting. 06/11/2017, Bing Li
81                 case MenuOptions.REGISTER_CHATTING:
82                     chatRegistryResponse = ChatPeerSingleton.PEER().registerChat(ChatMaintainer.PEER()
83                         .getLocalUserKey(), ChatMaintainer.PEER().getLocalUsername(), ChatMaintainer.PEER()
84                         .getLocalUsername() + " is a good guy", "Computer science is interesting?");
85                     System.out.println("Chatting registry status: " + chatRegistryResponse.isSucceeded());
86                     break;
87
88                 // Search the potential chatting partner. 06/11/2017, Bing Li
89                 case MenuOptions.SEARCH_USER:
90                     chatPartnerResponse = ChatPeerSingleton.PEER().searchUser(ChatMaintainer.PEER()
91                         .getPartnerKey());
92                     System.out.println(chatPartnerResponse.getUserName() + ": " + chatPartnerResponse
```

```
93                      .getDescription());
94                  System.out.println(chatPartnerResponse.getUserName() + ": " + chatPartnerResponse
95                      .getPreference());
96                  System.out.println(chatPartnerResponse.getUserName() + " is located at " +
97                      chatPartnerResponse.getIP() + ":" + chatPartnerResponse.getPort());
98                  ChatMaintainer.PEER().setPartnerIP(chatPartnerResponse.getIP());
99                  ChatMaintainer.PEER().setPartnerPort(chatPartnerResponse.getPort());
100                 break;
101
102             // Add the partner for chatting. 06/11/2017, Bing Li
103             case MenuOptions.ADD_FRIEND:
104                 if (ChatMaintainer.PEER().getPartnerIP() != null)
105                 {
106                     ChatPeerSingleton.PEER().notifyAddFriend(ChatMaintainer.PEER().getPartnerIP(),
107                         ChatMaintainer.PEER().getPartnerPort());
108                 }
109                 else
110                 {
111                     System.out.println("You need to add search your partner, " + ChatMaintainer.PEER()
112                         .getPartner() + ", first before add her as a friend!");
113                 }
114                 break;
115
116             // Start to chat. 06/11/2017, Bing Li
117             case MenuOptions.START_CHATTING:
118                 int chatOption = ChatOptions.NO_OPTION;
119                 String optionStr;
120                 // Present a sub menu for chatting. 06/11/2017, Bing Li
121                 while (chatOption != ChatOptions.QUIT_CHAT)
122                 {
123                     ChatUI.PEER().printMenu();
124                     optionStr = in.nextLine();
125                     try
126                     {
127                         chatOption = Integer.parseInt(optionStr);
128                         System.out.println("Your choice: " + option);
129                         ChatUI.PEER().sent(chatOption);
130                     }
131                     catch (NumberFormatException e)
132                     {
133                         chatOption = MenuOptions.NO_OPTION;
134                         System.out.println(ClientMenu.WRONG_OPTION);
135                     }
136                 }
137                 break;
138
139             case MenuOptions.QUIT:
140                 break;
141         }
142     }
143 }
```

List 6.55 The code of ClientUI.java

The code of ClientUI in List 6.55 has the similar structure to the one in List 5.46. But since they are not related to distributed programming, I do not want to give it any definition. It is also straightforward. Three of the four client methods in List 6.43 are called by the code at Line 82 (registerChat()), Line 90 (searchUser()) and Line 106 (notifyAddFriend()), respectively.

Another UI, ChatUI, is shown in List 6.56. The UI invokes the method, notifyChat(), at Line 72.

```
1   package com.greatfree.chat.peer;
2
3   import java.util.Scanner;
4
5   import com.greatfree.chat.ChatMenu;
6   import com.greatfree.chat.ChatOptions;
7
```

```
8       /*
9        * The class aims to print a menu list on the screen for users to interact with the client and chat with the
10       * chatting peer. The menu is unique in the client such that it is implemented in the pattern of
11       * a singleton. 04/27/2017, Bing Li
12       */
13
14      // Created: 05/02/2017, Bing Li
15      public class ChatUI
16      {
17          private Scanner in = new Scanner(System.in);
18
19          /*
20           * Initialize. 04/27/2017, Bing Li
21           */
22          private ChatUI()
23          {
24          }
25
26          /*
27           * Initialize a singleton. 04/23/2017, Bing Li
28           */
29          private static ChatUI instance = new ChatUI();
30
31          public static ChatUI PEER()
32          {
33              if (instance == null)
34              {
35                  instance = new ChatUI();
36                  return instance;
37              }
38              else
39              {
40                  return instance;
41              }
42          }
43
44          public void dispose()
45          {
46              this.in.close();
47          }
48
49          /*
50           * Print the menu list on the screen. 04/23/2017, Bing Li
51           */
52          public void printMenu()
53          {
54              System.out.println(ChatMenu.MENU_HEAD);
55              System.out.println(ChatMenu.TYPE_MESSAGE + ChatMaintainer.PEER().getPartner());
56              System.out.println(ChatMenu.QUIT);
57              System.out.println(ChatMenu.MENU_TAIL);
58              System.out.println(ChatMenu.INPUT_PROMPT);
59          }
60
61          /*
62           * Send messages to the chatting peer. 04/23/2017, Bing Li
63           */
64          public void sent(int option)
65          {
66              switch (option)
67              {
68                  // Send the chatting message to the chatting partner. 06/11/2017, Bing Li
69                  case ChatOptions.TYPE_CHAT:
70                      System.out.println("Please type your message: ");
71                      String message = in.nextLine();
72                      ChatPeerSingleton.PEER().notifyChat(ChatMaintainer.PEER().getPartnerIP(),
73                          ChatMaintainer.PEER().getPartnerPort(), message);
74                      break;
75
76                  case ChatOptions.QUIT_CHAT:
77                  break;
78              }
79          }
80      }
```

List 6.56 The code of ChatUI.java

5. Testing

Now all of the code for the peer-based chatting application has been elaborated. We need to test the system to verify whether GreatFree APIs and patterns work or not. To be more practical, I prepare for four PCs to test the system. You can see the four figures, Figure 6.1, Figure 6.2, Figure 6.3 and Figure 6.4, each of which has its own IP addresses.

```
imac — libing@freescape: ~ — ssh — 80×40
greatfree@Mum.local: ~$ cd /GreatFreeLabs/Clouds/
greatfree@Mum.local: /GreatFreeLabs/Clouds$ ls
Administrator   ChatPeer        Client          ClusterRoot
ChatAdmin       ChatRegistry    Cluster         Server
ChatClient      ChatServer      ClusterChild
greatfree@Mum.local: /GreatFreeLabs/Clouds$ cd ChatRegistry/
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatRegistry$ ls
build           build.xml       lib             src
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatRegistry$ ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
        options=3<RXCSUM,TXCSUM>
        inet6 ::1 prefixlen 128
        inet 127.0.0.1 netmask 0xff000000
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
        nd6 options=1<PERFORMNUD>
gif0: flags=8010<POINTOPOINT,MULTICAST> mtu 1280
stf0: flags=0<> mtu 1280
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        options=27<RXCSUM,TXCSUM,VLAN_MTU,TSO4>
        ether 00:26:b0:ed:f2:10
        nd6 options=1<PERFORMNUD>
        media: autoselect
        status: inactive
en1: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        ether f8:1e:df:df:91:0d
        inet6 fe80::fa1e:dfff:fedf:910d%en1 prefixlen 64 scopeid 0x5
        inet 192.168.1.107 netmask 0xffffff00 broadcast 192.168.1.255
        nd6 options=1<PERFORMNUD>
        media: autoselect
        status: active
fw0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 4078
        lladdr 00:26:b0:ff:fe:ed:f2:10
        nd6 options=1<PERFORMNUD>
        media: autoselect <full-duplex>
        status: inactive
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
        ether 0a:1e:df:df:91:0d
        media: autoselect
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatRegistry$
```

Figure 6.1 The terminal for the registry server is ready

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
          inet addr:192.168.1.109  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::28d0:31a:338:81a4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:5067 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2733 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:5849839 (5.8 MB)  TX bytes:219012 (219.0 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:267 errors:0 dropped:0 overruns:0 frame:0
          TX packets:267 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:21230 (21.2 KB)  TX bytes:21230 (21.2 KB)

libing@mother:~$ cd GreatFreeLabs/Clouds/ChatPeer/
libing@mother:~/GreatFreeLabs/Clouds/ChatPeer$ ls
build  build.xml  lib  src
libing@mother:~/GreatFreeLabs/Clouds/ChatPeer$ ifconfig
eth0      Link encap:Ethernet  HWaddr 9c:5c:8e:bf:50:6b
          inet addr:192.168.1.109  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::28d0:31a:338:81a4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8124 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4328 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:9154944 (9.1 MB)  TX bytes:421215 (421.2 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:267 errors:0 dropped:0 overruns:0 frame:0
          TX packets:267 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:21230 (21.2 KB)  TX bytes:21230 (21.2 KB)

libing@mother:~/GreatFreeLabs/Clouds/ChatPeer$
```

Figure 6.2 One chatting peer terminal is ready

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40

 * Documentation:  https://help.ubuntu.com/

New release '14.04.4 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


Your Hardware Enablement Stack (HWE) is supported until April 2017.

Last login: Sun Jun 11 23:41:00 2017 from labs.local
libing@freescape:~$ cd GreatFreeLabs/Clouds/ChatPeer/
libing@freescape:~/GreatFreeLabs/Clouds/ChatPeer$ ifconfig
eth0      Link encap:Ethernet  HWaddr f0:de:f1:45:df:be
          inet addr:192.168.1.110  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::f2de:f1ff:fe45:dfbe/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:5646 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3432 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3868329 (3.8 MB)  TX bytes:443608 (443.6 KB)
          Interrupt:20 Memory:f2600000-f2620000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:206 errors:0 dropped:0 overruns:0 frame:0
          TX packets:206 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:16885 (16.8 KB)  TX bytes:16885 (16.8 KB)

wlan0     Link encap:Ethernet  HWaddr 58:94:6b:a5:05:a4
          inet6 addr: fe80::5a94:6bff:fea5:5a4/64 Scope:Link
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:125 errors:0 dropped:0 overruns:0 frame:0
          TX packets:49 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:15653 (15.6 KB)  TX bytes:10551 (10.5 KB)

libing@freescape:~/GreatFreeLabs/Clouds/ChatPeer$
```

Figure 6.3 Another chatting peer terminal is ready

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatAdmin — ssh — 85×40
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:104 errors:0 dropped:0 overruns:0 frame:0
          TX packets:104 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:8381 (8.3 KB)  TX bytes:8381 (8.3 KB)

libing@freeweb:~$ cd GreatFreeLabs/Clouds/
libing@freeweb:~/GreatFreeLabs/Clouds$ ls
Administrator  ChatClient  ChatRegistry  Client   ClusterChild  Server
ChatAdmin      ChatPeer    ChatServer    Cluster  ClusterRoot
libing@freeweb:~/GreatFreeLabs/Clouds$ cd ChatAdmin/
libing@freeweb:~/GreatFreeLabs/Clouds/ChatAdmin$ ls
build  build.xml  lib  src
libing@freeweb:~/GreatFreeLabs/Clouds/ChatAdmin$ ifconfig
eth0      Link encap:Ethernet  HWaddr b8:ac:6f:92:ef:ef
          inet addr:192.168.1.112  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::baac:6fff:fe92:efef/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:5378 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3474 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3840316 (3.8 MB)  TX bytes:433200 (433.2 KB)

eth1      Link encap:Ethernet  HWaddr b8:ac:6f:92:ef:f0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:136 errors:0 dropped:0 overruns:0 frame:0
          TX packets:136 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:11211 (11.2 KB)  TX bytes:11211 (11.2 KB)

libing@freeweb:~/GreatFreeLabs/Clouds/ChatAdmin$
```

Figure 6.4 The terminal for the chatting administrator is ready

5.1 Starting the Registry Server

Before we start the testing, we should know that the IP address of the registry server must be known to all of the computers within the system. The value is set as a constant, CHAT_REGISTRY_ADDRESS, at Line 13 in ChatConfig.java in List 5.56. In my case, the registry server is located at the IP, 192.168.1.107. So you need to update the constant of CHAT_REGISTRY_ADDRESS to the value.

To test the chatting application on the P2P distributed model, the first step is to start up the registry server. We have indicated the importance of the registry server. After deploying the registry server to Ant, you can start it by typing the command, ant, as shown in Figure 6.5.

```
imac — libing@freescape: ~ — ssh — 80×40
        status: inactive
en1: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        ether f8:1e:df:df:91:0d
        inet6 fe80::fa1e:dfff:fedf:910d%en1 prefixlen 64 scopeid 0x5
        inet 192.168.1.107 netmask 0xffffff00 broadcast 192.168.1.255
        nd6 options=1<PERFORMNUD>
        media: autoselect
        status: active
fw0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 4078
        lladdr 00:26:b0:ff:fe:ed:f2:10
        nd6 options=1<PERFORMNUD>
        media: autoselect <full-duplex>
        status: inactive
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
        ether 0a:1e:df:df:91:0d
        media: autoselect
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatRegistry$ ant
Buildfile: /GreatFreeLabs/Clouds/ChatRegistry/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ChatRegistry/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ChatRegistry/bui
ld/classes

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ChatRegistry/build/jar/FreeWeb.j
ar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
```

Figure 6.5 The registry server is started

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
libing@mother:~/GreatFreeLabs/Clouds/ChatPeer$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ChatPeer/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ChatPeer/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatPeer/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatPeer/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatPeer/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ChatPeer/
build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatPeer/build/jar/FreeWe
b.jar

run:
     [java] Tell me your user name:
gates
     [java] Tell me your partner:
greatfree
     [java] Chatting peer starting up ...
     [java]
     [java] super.getPort() = 8944
     [java] response.getPeerPort() = 8944
     [java] Chatting peer started ...
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 6.6 Start the peer of gates up

5.2 Starting Chatting Peers

Now we can start the chatting peers. Similar to the testing for the C/S chatting application, we call one peer, gates, and another peer, greatfree.

```
imac — libing@freescape: ~ — ssh — 80×40
        nd6 options=1<PERFORMNUD>
        media: autoselect
        status: active
fw0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 4078
        lladdr 00:26:b0:ff:fe:ed:f2:10
        nd6 options=1<PERFORMNUD>
        media: autoselect <full-duplex>
        status: inactive
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
        ether 0a:1e:df:df:91:0d
        media: autoselect
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatRegistry$ ant
Buildfile: /GreatFreeLabs/Clouds/ChatRegistry/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ChatRegistry/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ChatRegistry/bui
ld/classes

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ChatRegistry/build/jar/FreeWeb.j
ar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
     [java] REGISTER_PEER_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] PORT_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] 1) getIdlePort(): ip = 192.168.1.109
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
```

Figure 6.7 The terminal of the registry server gets updates

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
libing@freescape:~/GreatFreeLabs/Clouds/ChatPeer$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ChatPeer/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ChatPeer/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatPeer/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatPeer/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatPeer/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ChatPeer/
build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatPeer/build/jar/FreeWe
b.jar

run:
     [java] Tell me your user name:
greatfree
     [java] Tell me your partner:
gates
     [java] Chatting peer starting up ...
     [java]
     [java] super.getPort() = 8944
     [java] response.getPeerPort() = 8944
     [java] Chatting peer started ...
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 6.8 The peer of greatfree is started up

First, start the peer of gates up, as shown in Figure 6.6. When the peer is executed, you need to specify the local user name, gates, and the partner user name, greatfree.

After the peer of gates is started up, the registry server gets some updates in its terminal, as shown in Figure 6.7. The requests, REGISTER_PEER_REQUEST and PORT_REQUEST, are received by the registry server. According to Figure 6.6 and Figure 6.7, it shows that the peer of gates gets one idle port from the registry server.

Next, let us start another peer, greatfree, as shown in Figure 6.8. You are required to specify the local user name as well as the partner name. You might feel weird why the port number is also 8944, which is the same as the one for the chatting peer of gates. The reason is simple. In the testing environment, each peer has its own PC. So their IP addresses are different. Then, even though the ports are the same, that is fine. If you do not have such a condition, the peers on the same PC must get different ports from the registry server. Figure 6.9 demonstrates the updates on the registry server after the peers are started up.

```
imac — libing@freescape: ~ — ssh — 80×40
        nd6 options=1<PERFORMNUD>
        media: autoselect <full-duplex>
        status: inactive
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
        ether 0a:1e:df:df:91:0d
        media: autoselect
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatRegistry$ ant
Buildfile: /GreatFreeLabs/Clouds/ChatRegistry/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ChatRegistry/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ChatRegistry/bui
ld/classes

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ChatRegistry/build/jar/FreeWeb.j
ar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
     [java] REGISTER_PEER_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] PORT_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] 1) getIdlePort(): ip = 192.168.1.109
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] REGISTER_PEER_REQUEST received @Mon Jun 12 00:45:14 PDT 2017
     [java] PORT_REQUEST received @Mon Jun 12 00:45:14 PDT 2017
     [java] 1) getIdlePort(): ip = 192.168.1.110
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
```

Figure 6.9 The registry server gets updates after two chatting peers are started up

5.3 Starting Up the Administrator

The last step is to start up the administrator. Actually, the administrator can be started at any time. As we discuss in Section 4 of Chapter 5, the administrator is a traditional client, which is made up with RR and RE patterns. So it does not need to interact with any others before it sends any commands. After it is started, the terminal is shown in Figure 6.10.

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatAdmin — ssh — 85×40

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:136 errors:0 dropped:0 overruns:0 frame:0
          TX packets:136 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:11211 (11.2 KB)  TX bytes:11211 (11.2 KB)

libing@freeweb:~/GreatFreeLabs/Clouds/ChatAdmin$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ChatAdmin
/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/jar/FreeW
eb.jar

run:
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Stop CS Chatting Server
     [java]     2) Stop Chatting Registry Server
     [java]     3) Stop Cluster Children
     [java]     0) End
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 6.10 The terminal of the administrator after it is started up

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatPeer/build/jar/FreeWe
b.jar

run:
     [java] Tell me your user name:
gates
     [java] Tell me your partner:
greatfree
     [java] Chatting peer starting up ...
     [java]
     [java] super.getPort() = 8944
     [java] response.getPeerPort() = 8944
     [java] Chatting peer started ...
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 1
     [java] Chatting registry status: true
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 6.11 The terminal of gates peer after the application-level registry is done

5.4 Register Chatting Users

When the two chatting peers are started up, they are registered on the registry server as the application-independent nodes. You can know that from the port assignments

and the updates on the registry server. However, before starting to chat, it still needs to register the additional application level registry of chatting.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatPeer/build/jar/FreeWe
b.jar

run:
     [java] Tell me your user name:
greatfree
     [java] Tell me your partner:
gates
     [java] Chatting peer starting up ...
     [java]
     [java] super.getPort() = 8944
     [java] response.getPeerPort() = 8944
     [java] Chatting peer started ...
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 1
     [java] Chatting registry status: true
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 6.12 The terminal of greatfree terminal after the application-level registry is done

```
imac — libing@freescape: ~ — ssh — 80×40
        status: inactive
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
        ether 0a:1e:df:df:91:0d
        media: autoselect
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatRegistry$ ant
Buildfile: /GreatFreeLabs/Clouds/ChatRegistry/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ChatRegistry/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ChatRegistry/bui
ld/classes

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ChatRegistry/build/jar/FreeWeb.j
ar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
     [java] REGISTER_PEER_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] PORT_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] 1) getIdlePort(): ip = 192.168.1.109
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] REGISTER_PEER_REQUEST received @Mon Jun 12 00:45:14 PDT 2017
     [java] PORT_REQUEST received @Mon Jun 12 00:45:14 PDT 2017
     [java] 1) getIdlePort(): ip = 192.168.1.110
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] PEER_CHAT_REGISTRY_REQUEST received @Mon Jun 12 01:30:44 PDT 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Mon Jun 12 01:45:06 PDT 2017
```

Figure 6.13 The registry server terminal after two peers register their application-related information

On the terminal of gates, you can select the first option, “1) Register Chatting: gates”, from the menu. Immediately after you make the selection, the registry server shows it receives one request, PEER_CHAT_REGISTRY_REQUEST. It represents that chatting dependent information is registered on the server.

Then, you need to do the same thing in terminal of greatfree. After you accomplish the task, the same thing happens on both of the chatting peer and the registry server. Figure 6.11, Figure 6.12 and Figure 6.13 demonstrate the peer of gates, the peer of greatfree and the registry server respectively after the application-level registry is done.

5.5 Searching Chatting Partners

After the application level information is registered, each peer is able to search its partners from the registry server. You can do that by choosing the second option from the menu. Until now, from the user’s point of view, it seems to be the same as the chatting application on the C/S distributed model. However, if you look at the registry server, you should detect the difference. All of the updates on the registry server are caused by your operations on the peers’ terminals. If you do nothing, the registry has no updates. That is different from the chatting server in Chapter 5. In that testing, once if a chatting client starts up, there are always polling requests periodically displayed on the chatting server.

Figure 6.14 illustrates the terminal of gates after the user of gates chooses the second option of search. Figure 6.15 shows the terminal of greatfree for the occurrence of the search. Figure 6.16 demonstrates the updates on the registry server. According to the terminal, the registry server receives two requests of PEER_CHAT_PARTNER_REQUEST.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
     [java] Chatting peer started ...
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 1
     [java] Chatting registry status: true
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice: 2
     [java] greatfree: greatfree is a good guy
     [java] greatfree: Computer science is interesting?
     [java] greatfree is located at 192.168.1.110:8944
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 6.14 The terminal of gates peer after searching its partner, greatfree

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
     [java] Chatting peer started ...
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 1
     [java] Chatting registry status: true
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice: 2
     [java] gates: gates is a good guy
     [java] gates: Computer science is interesting?
     [java] gates is located at 192.168.1.109:8944
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 6.15 The terminal of greatfree after searching its partner, gates

```
imac — libing@freescape: ~ — ssh — 80×40
        ether 0a:1e:df:df:91:0d
        media: autoselect
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatRegistry$ ant
Buildfile: /GreatFreeLabs/Clouds/ChatRegistry/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ChatRegistry/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ChatRegistry/bui
ld/classes

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ChatRegistry/build/jar/FreeWeb.j
ar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
     [java] REGISTER_PEER_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] PORT_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] 1) getIdlePort(): ip = 192.168.1.109
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] REGISTER_PEER_REQUEST received @Mon Jun 12 00:45:14 PDT 2017
     [java] PORT_REQUEST received @Mon Jun 12 00:45:14 PDT 2017
     [java] 1) getIdlePort(): ip = 192.168.1.110
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] PEER_CHAT_REGISTRY_REQUEST received @Mon Jun 12 01:30:44 PDT 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Mon Jun 12 01:45:06 PDT 2017
     [java] PEER_CHAT_PARTNER_REQUEST received @Mon Jun 12 02:58:49 PDT 2017
     [java] PEER_CHAT_PARTNER_REQUEST received @Mon Jun 12 02:58:53 PDT 2017
```

Figure 6.16 The terminal of the registry server after two partner searching requests are received

5.6 Adding Chatting Partners

Before a chatting is started, a user needs to send an invitation to its potential partner. This is what the third option in the menu means. If you do that from the terminal of gates, you must notice some differences again from the C/S chatting system.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
     [java] Your choice: 1
     [java] Chatting registry status: true
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice: 2
     [java] greatfree: greatfree is a good guy
     [java] greatfree: Computer science is interesting?
     [java] greatfree is located at 192.168.1.110:8944
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
3
     [java] Your choice: 3
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 6.17 The terminal of gates selects the third option, Add Friend: greatfree

```
imac — libing@freescape: ~ — ssh — 80×40
        ether 0a:1e:df:df:91:0d
        media: autoselect
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatRegistry$ ant
Buildfile: /GreatFreeLabs/Clouds/ChatRegistry/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ChatRegistry/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ChatRegistry/bui
ld/classes

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ChatRegistry/build/jar/FreeWeb.j
ar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
     [java] REGISTER_PEER_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] PORT_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] 1) getIdlePort(): ip = 192.168.1.109
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] REGISTER_PEER_REQUEST received @Mon Jun 12 00:45:14 PDT 2017
     [java] PORT_REQUEST received @Mon Jun 12 00:45:14 PDT 2017
     [java] 1) getIdlePort(): ip = 192.168.1.110
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] PEER_CHAT_REGISTRY_REQUEST received @Mon Jun 12 01:30:44 PDT 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Mon Jun 12 01:45:06 PDT 2017
     [java] PEER_CHAT_PARTNER_REQUEST received @Mon Jun 12 02:58:49 PDT 2017
     [java] PEER_CHAT_PARTNER_REQUEST received @Mon Jun 12 02:58:53 PDT 2017
```

Figure 6.18 The registry server has no updates after any users select the option of adding partners

After selecting the third option in the menu on the terminal of gates, as shown in Figure 6.17, no any updates are found on the registry server. However, the terminal of greatfree updates simultaneously. It receives a message that says “I want to chat with you!”. The reason for that is that the peer of gates has got the IP address of its partner, greatfree, after searching greatfree in the second step. So gates is able to send messages to greatfree directly without sending polling requests to any brokers.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 1
     [java] Chatting registry status: true
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice: 2
     [java] gates: gates is a good guy
     [java] gates: Computer science is interesting?
     [java] gates is located at 192.168.1.109:8944
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PEER_ADD_PARTNER_NOTIFICATION received @Mon Jun 12 03:21:47 CST 2017
     [java] gates says: Hello, I want to chat with you!
```

Figure 6.19 The terminal of greatfree after its partner selects the option, Add Friend: greatfree

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice: 2
     [java] gates: gates is a good guy
     [java] gates: Computer science is interesting?
     [java] gates is located at 192.168.1.109:8944
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PEER_ADD_PARTNER_NOTIFICATION received @Mon Jun 12 03:21:47 CST 2017
     [java] gates says: Hello, I want to chat with you!
3
     [java] Your choice: 3
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 6.20 The terminal of greatfree selects the third option, Add Friend: gates

You can do the same thing at the terminal of greatfree, as shown in Figure 6.20. Immediately after that, the terminal of gates receives the message, “I want to chat with you!”, as shown in Figure 6.21. At the same time, the terminal of the registry server has no any updates, as shown in Figure 6.18.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice: 2
     [java] greatfree: greatfree is a good guy
     [java] greatfree: Computer science is interesting?
     [java] greatfree is located at 192.168.1.110:8944
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
3
     [java] Your choice: 3
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PEER_ADD_PARTNER_NOTIFICATION received @Mon Jun 12 03:36:15 CST 2017
     [java] greatfree says: Hello, I want to chat with you!
```

Figure 6.21 The terminal of gates receives a chatting invitation immediately after greatfree adds gates

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
3
     [java] Your choice: 3
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PEER_ADD_PARTNER_NOTIFICATION received @Mon Jun 12 03:36:15 CST 2017
     [java] greatfree says: Hello, I want to chat with you!
4
     [java] Your choice: 4
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 4
     [java] Please type your message:
hello, greatfree
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 6.22 Gates types, “hello, greatfree”, to greatfree

5.7 Chatting

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice: 2
     [java] gates: gates is a good guy
     [java] gates: Computer science is interesting?
     [java] gates is located at 192.168.1.109:8944
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PEER_ADD_PARTNER_NOTIFICATION received @Mon Jun 12 03:21:47 CST 2017
     [java] gates says: Hello, I want to chat with you!
3
     [java] Your choice: 3
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PEER_CHAT_NOTIFICATION received @Mon Jun 12 03:44:22 CST 2017
     [java] gates says, hello, greatfree
```

Figure 6.23 Greatfree receives, “hello, greatfree”, from gates

```
imac — libing@freescape: ~ — ssh — 80×40
        ether 0a:1e:df:df:91:0d
        media: autoselect
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatRegistry$ ant
Buildfile: /GreatFreeLabs/Clouds/ChatRegistry/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ChatRegistry/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ChatRegistry/bui
ld/classes

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ChatRegistry/build/jar/FreeWeb.j
ar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
     [java] REGISTER_PEER_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] PORT_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] 1) getIdlePort(): ip = 192.168.1.109
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] REGISTER_PEER_REQUEST received @Mon Jun 12 00:45:14 PDT 2017
     [java] PORT_REQUEST received @Mon Jun 12 00:45:14 PDT 2017
     [java] 1) getIdlePort(): ip = 192.168.1.110
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] PEER_CHAT_REGISTRY_REQUEST received @Mon Jun 12 01:30:44 PDT 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Mon Jun 12 01:45:06 PDT 2017
     [java] PEER_CHAT_PARTNER_REQUEST received @Mon Jun 12 02:58:49 PDT 2017
     [java] PEER_CHAT_PARTNER_REQUEST received @Mon Jun 12 02:58:53 PDT 2017
```

Figure 6.24 The registry has no updates during the chatting procedure

Now everything is ready for chatting. We have taken a long journey to reach the step. You can choose the fourth option from the menu. Then, a submenu is displayed. You just select the first one, “Type Message To: greatfree”. Thereafter, you are prompted to type chatting message. For example, you can type, “hello, greatfree”, as shown in Figure 6.22. Simultaneously, the terminal of greatfree receives your message, “hello,

greatfree", as shown in Figure 6.23. Well, the registry server still has no any updates, as shown in Figure 6.24. Again, no polling requests are displayed in any terminals.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PEER_CHAT_NOTIFICATION received @Mon Jun 12 03:44:22 CST 2017
     [java] gates says, hello, greatfree
4
     [java] Your choice: 4
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 4
     [java] Please type your message:
yes? I am fine
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PEER_CHAT_NOTIFICATION received @Mon Jun 12 04:02:36 CST 2017
     [java] gates says, do you like the class of cloud computing?
1
     [java] Your choice: 4
     [java] Please type your message:
no, it's boring
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 6.25 Greatfree types some messages to gates

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
     [java]
     [java] Input an option:
     [java] PEER_ADD_PARTNER_NOTIFICATION received @Mon Jun 12 03:36:15 CST 2017
     [java] greatfree says: Hello, I want to chat with you!
4
     [java] Your choice: 4
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 4
     [java] Please type your message:
hello, greatfree
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PEER_CHAT_NOTIFICATION received @Mon Jun 12 04:01:23 CST 2017
     [java] greatfree says, yes? I am fine
1
     [java] Your choice: 4
     [java] Please type your message:
do you like the class of cloud computing?
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PEER_CHAT_NOTIFICATION received @Mon Jun 12 04:02:45 CST 2017
     [java] greatfree says, no, it's boring
```

Figure 6.26 Gates chats with greatfree

You must have understood the differences from the C/S chatting system, which we discuss in Chapter 5. Since the peers can interact with each other, their chatting messages can be transmitted between them instantly if the latency between them is low enough. For the testing, it is performed in a LAN. So the message can be transmitted with a very low latency, which chatting users cannot be aware.

Correspondingly, greatfree can also type messages at its terminal, as shown in Figure 6.25. After that, gates must be able to receive the messages immediately, as shown Figure 6.26. Again, the registry server always has no any updates. In this way, you can test with more messages.

5.8 Terminating the System

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
     [java]     1) Type Message To: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PEER_CHAT_NOTIFICATION received @Mon Jun 12 04:01:23 CST 2017
     [java] greatfree says, yes? I am fine
1
     [java] Your choice: 4
     [java] Please type your message:
do you like the class of cloud computing?
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PEER_CHAT_NOTIFICATION received @Mon Jun 12 04:02:45 CST 2017
     [java] greatfree says, no, it's boring
0
     [java] Your choice: 4
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: gates
     [java]     2) Search User: greatfree
     [java]     3) Add Friend: greatfree
     [java]     4) Start Chatting with: greatfree
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
0
     [java] Your choice: 0

main:

BUILD SUCCESSFUL
Total time: 233 minutes 42 seconds
libing@mother:~/GreatFreeLabs/Clouds/ChatPeer$
```

Figure 6.27 The chatting peer, gates, is closed

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ChatPeer — ssh — 85×40
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] PEER_CHAT_NOTIFICATION received @Mon Jun 12 04:02:36 CST 2017
     [java] gates says, do you like the class of cloud computing?
1
     [java] Your choice: 4
     [java] Please type your message:
no, it's boring
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
0
     [java] Your choice: 4
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Register Chatting: greatfree
     [java]     2) Search User: gates
     [java]     3) Add Friend: gates
     [java]     4) Start Chatting with: gates
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
0
     [java] Your choice: 0

main:

BUILD SUCCESSFUL
Total time: 214 minutes 29 seconds
libing@freescape:~/GreatFreeLabs/Clouds/ChatPeer$
```

Figure 6.28 The chatting peer, greatfree, is closed

The larger a system, the more difficult to manage. One indicator whether your system is well managed or not is whether it can be closed properly.

First, we can close the chatting peers by selecting the options of the menu to quit from the chatting peer. Although the peer can be terminated remotely. I suggest you to do that by its menu options in a higher priority. After the quit options are selected, the two peers are shut down properly, as shown in Figure 6.27 and Figure 6.28, respectively. Since each peer needs to unregister itself before it is shut down, the registry server receives two requests from the two peers, shown in Figure 6.29.

```
imac — libing@freescape: ~ — ssh — 80×40
clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ChatRegistry/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ChatRegistry/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ChatRegistry/bui
ld/classes

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ChatRegistry/build/jar/FreeWeb.j
ar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
     [java] REGISTER_PEER_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] PORT_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] 1) getIdlePort(): ip = 192.168.1.109
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] REGISTER_PEER_REQUEST received @Mon Jun 12 00:45:14 PDT 2017
     [java] PORT_REQUEST received @Mon Jun 12 00:45:14 PDT 2017
     [java] 1) getIdlePort(): ip = 192.168.1.110
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] PEER_CHAT_REGISTRY_REQUEST received @Mon Jun 12 01:30:44 PDT 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Mon Jun 12 01:45:06 PDT 2017
     [java] PEER_CHAT_PARTNER_REQUEST received @Mon Jun 12 02:58:49 PDT 2017
     [java] PEER_CHAT_PARTNER_REQUEST received @Mon Jun 12 02:58:53 PDT 2017
     [java] UNREGISTER_PEER_REQUEST received @Mon Jun 12 04:19:11 PDT 2017
     [java] removed port = 8944
     [java] removed port = 8943
     [java] UNREGISTER_PEER_REQUEST received @Mon Jun 12 04:19:23 PDT 2017
     [java] removed port = 8944
     [java] removed port = 8943
```

Figure 6.29 The registry server receives the requests for unregistering

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatAdmin — ssh — 85×40
clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ChatAdmin
/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/jar/FreeW
eb.jar

run:
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Stop CS Chatting Server
     [java]     2) Stop Chatting Registry Server
     [java]     3) Stop Cluster Children
     [java]     0) End
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice is: 2
     [java]     2) Stop Chatting Registry Server
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Stop CS Chatting Server
     [java]     2) Stop Chatting Registry Server
     [java]     3) Stop Cluster Children
     [java]     0) End
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 6.30 The administrator selects the second option, “Stop Chatting Registry Server”

There is one last step to finish the testing. To terminate the registry server, you have to get help from the administrator, as shown in Figure 6.10. The menu of the administrator has one option, “2) Stop Chatting Registry Server”. Just select it as shown in Figure 6.30. After that, the registry server receives one notification, SHUTDOWN_CHAT_SERVRE_NOTIFICATION, as shown in Figure 6.31. The notification causes the registry server being shut down.

```
imac — libing@freescape: ~ — ssh — 80×40

compile:
     [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ChatRegistry/bui
ld/classes

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ChatRegistry/build/jar/FreeWeb.j
ar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
     [java] REGISTER_PEER_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] PORT_REQUEST received @Mon Jun 12 00:25:50 PDT 2017
     [java] 1) getIdlePort(): ip = 192.168.1.109
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] REGISTER_PEER_REQUEST received @Mon Jun 12 00:45:14 PDT 2017
     [java] PORT_REQUEST received @Mon Jun 12 00:45:14 PDT 2017
     [java] 1) getIdlePort(): ip = 192.168.1.110
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] PEER_CHAT_REGISTRY_REQUEST received @Mon Jun 12 01:30:44 PDT 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Mon Jun 12 01:45:06 PDT 2017
     [java] PEER_CHAT_PARTNER_REQUEST received @Mon Jun 12 02:58:49 PDT 2017
     [java] PEER_CHAT_PARTNER_REQUEST received @Mon Jun 12 02:58:53 PDT 2017
     [java] UNREGISTER_PEER_REQUEST received @Mon Jun 12 04:19:11 PDT 2017
     [java] removed port = 8944
     [java] removed port = 8943
     [java] UNREGISTER_PEER_REQUEST received @Mon Jun 12 04:19:23 PDT 2017
     [java] removed port = 8944
     [java] removed port = 8943
     [java] SHUTDOWN_CHAT_SERVER_NOTIFICATION received @Mon Jun 12 04:31:04 PDT
2017

main:

BUILD SUCCESSFUL
Total time: 278 minutes 50 seconds
greatfree@Mum.local: /GreatFreeLabs/Clouds/ChatRegistry$
```

Figure 6.31 The registry server is shut down after receiving the notification, SHUTDOWN_CHAT_SERVER_NOTIFICATION

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatAdmin — ssh — 85×40
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ChatAdmin
/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/jar/FreeW
eb.jar

run:
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Stop CS Chatting Server
     [java]     2) Stop Chatting Registry Server
     [java]     3) Stop Cluster Children
     [java]     0) End
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice is: 2
     [java]     2) Stop Chatting Registry Server
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Stop CS Chatting Server
     [java]     2) Stop Chatting Registry Server
     [java]     3) Stop Cluster Children
     [java]     0) End
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
0
     [java] Your choice is: 0

main:

BUILD SUCCESSFUL
Total time: 216 minutes 50 seconds
libing@freeweb:~/GreatFreeLabs/Clouds/ChatAdmin$
```

Figure 6.32 The administrator is closed by selecting the End option

6. A Summary

Now you have become a senior developer in the domain of distributed programming and cloud programming. Congratulations! Although this chapter we program a chatting application based on a different distributed model, P2P, rather than the one of C/S, we notice that the techniques we employ are still those we have learned in the previous chapters, such as the APIs and the patterns. Most of our previous experiences can be reused in the new application. In addition, at least, when programming distributed techniques, your behaviors are still CPR only. That represents one success of GreatFree, distributed programming techniques are highly patterned. Plus effective APIs, programming distributed systems with GreatFree is straightforward and comfortable.

Until now, we have programming some distributed structures, such as C/S, C/S chatting and P2P chatting. Each of them represents one type of fundamental distributed applications. The one of C/S is the application that is performed between two nodes only. The C/S chatting fulfills the requirement if multiple clients need to interact with each other with the C/S distributed model. The P2P chatting provides direct interactions between distributed nodes. With the code and the infrastructures, developers can place additional code on them with simplified behaviors, i.e., CPR.

Although the distributed structures we have talked about can solve many requirements in practice, it does not cover all of them. At least, the previous ones perform interactions between the mode of two individual distributed nodes. Is it possible to carry out interactions among a large number of computers over the Internet. For example, one node is able to send messages to an unlimited number of others? This model is very useful as well in many cases. If you know the solution, you can program more complicated distributed applications.

Chapter 7 Programming with Cluster

Abstract

Cluster programming is an extremely tough challenge for developers, especially when they have no any frameworks to assist. According to my experiences, I have never seen any programming languages or environments to support that. In any cases, developers have to borrow mature frameworks to handle such a difficult problem. Fortunately, GreatFree intends to take the challenge. Until now, it does a good job in the sense that developers are offered a bunch of APIs and patterns to construct an unlimited scale cluster through programming without borrowing any third party distributed frameworks.

You might wonder what is a cluster. As a book to introduce a practical programming environment rather than an academic paper, I do not propose a rigorous definition. Luckily, this is a concept that is easy to understand. A cluster is an aggregation of multiple computers to work in the way like a single one to deal with much heavier workload that each one in the system cannot process by itself. Simply speaking, it is necessary to integrate a certain number of computers when the computing task is big. It often happens in the current computing environment based on the Internet. Any computing systems that aim to serve users as many as possible should encounter such a problem. If you happen to be a rich man, you can buy high performance computers to solve the problem. If not, for example, if you are just an entrepreneur to start up your own company, every penny is so important to your next step. Then, it is impossible for you to afford high cost. In this case, you can still have another solution. At this time, there are so many commodity computers or used ones in the market. If you search the keyword, used computers, on amazon.com or any other e-commercial Web sites, you can get thousands of computers to be sold. Although each one of them is not powerful, you can weave them together to form a strong computing system. It saves you money as well as accomplish your tasks. That must be a good idea, right?

Actually, the initiators of Google ever did that at the beginning of the giant. The technique, GFS [1], they employed on this keeps alive till now although it is upgraded deeply. But I would like to say that GFS is a framework that is dedicated to establish the distributed file system. If your task is different, for example, if you plan to send one subtask to each computer such that you can get the merged result or if you come up with a huge memory pool that is made up with those cheap computers, is it possible to do that with GFS or is it always feasible to do that with similar frameworks?

From my point of view, each distributed framework is usually designed for one particular scenario. If you can find the one that meets your requirements, that is fine since it must speed up your development speed. If not, what is your solution? Or if your requirements cannot be exactly solved, can you make some changes? Or even if one framework is suitable, but it is expensive, how can you solve the problem?

It seems that we talk about a lot of issues that are not related to programming with GreatFree. From a programmer's point of view, I would rather ask why no programming language helps us accomplish this goal. At least, GreatFree provides developers with such functions with flexibility. Because of that, you can implement any types of clusters to meet your specific requirements rapidly through programming without any additional costs.

1. The APIs for Clustering

If GreatFree is so amazing like what I have presented, what do those APIs look like? In the past chapters, many APIs have shown you. The primary difference between the ones in the chapter and the ones in the previous chapters is that we need to do something to allow developers to program the interactions among multiple distributed nodes instead of only the manner for one-to-one, such as one client to one server and one peer to one another. Using the APIs and patterns in the chapter, developers can send notifications to an infinite number of distributed nodes. In the same way, a request can also be sent to unlimited nodes as well as obtaining responses from all or any of them. Thus, it is possible to perform distributed computing in a large-scale simultaneously rather than just between two nodes. Furthermore, the large number of nodes is virtualized as a single powerful computing node, a client, a server or a peer.

Message Type	Multicasting Type	APIs for Root	APIs for Children
Notification	Notification Broadcasting	ClusterRootBroadcastNotifier	ClusterChildBroadcastNotifier
	Notification Anycasting	ClusterRootAnycastNotifier	N/A
	Notification Unicasting	ClusterRootAnycastNotifier	N/A
Request/Response	Request Broadcasting	ClusterRootBroadcastReader	ClusterChildBroadcastReader
	Request Anycasting	ClusterRootAnycastReader	N/A
	Request Unicasting	ClusterRootUnicastReader	N/A

Table 7.1 The multicasting APIs of GreatFree

1.1 The Types of Multicasting

Before we start to talk about clustering programming, we need to learn the knowledge about multicasting. In other words, a cluster is constructed on the techniques of multicasting. Multicasting indicates the technology that is employed to send messages from a single distributed node to multiple ones in a networked environment. At this time, it specifies the manner in which each node is connected through the protocol of TCP/IP and any one of the nodes is able to send messages to all, any, or only one of them. So, the multicasting can be divided into three fundamental approaches, broadcasting, anycasting or unicasting. In addition, because the messages are classified as the two types, i.e., notifications and requests, the multicasting in practice is categorized into six types, i.e., notification broadcasting, notification anycasting, notification unicasting, request broadcasting, request anycasting and request unicasting. Moreover, in the real world of programming, multicasting has different participants who play different roles during the procedure. In practice, the roles can be classified as the root and the children. Within the roles, the root is responsible for starting up one multicasting. That is, the initial message is sent from the root and the children are located at the hierarchical architecture of the multicasting to receive those messages. To handle the case, it is necessary to design different APIs for those different types of nodes.

1.2 The Multicasting APIs

For the above considerations, the number of APIs for the cases becomes larger in GreatFree. Table 7.1 presents the classification of the multicasting and relevant APIs supported by GreatFree. In the table, the APIs are presented with the class names only. Table 7.2 presents the constructors and the methods of those APIs.

API	Constructor	Method
ClusterRootBroadcastNotifier	ClusterRootBroadcastNotifier(FreeClientPool clientPool, int multicastorPoolSize, long notifierAssignWaitTime, NotificationCreator creator)	void dispose()
		void notify(Data data, int rootBranchCount, int subBranchCount)
ClusterRootAnycastNotifier	ClusterRootAnycastNotifier(FreeClientPool clientPool, int multicastorPoolSize, long notifierAssignWaitTime, NotificationCreator creator)	void dispose()
		void notify(Data data, int rootBranchCount, int subBranchCount)
ClusterRootUnicastNotifier	ClusterRootUnicastNotifier(FreeClientPool clientPool, int multicastorPoolSize, long notifierAssignWaitTime, NotificationCreator creator)	void dispose()
		void randomNotify(Data data, int rootBranchCount, int subBranchCount)
		void notify(Data data, int rootBranchCount, int subBranchCount, String childKey)
ClusterRootBroadcastReader	ClusterRootBroadcastReader(FreeClientPool clientPool, int multicastorPoolSize, long readerAssignWaitTime, long broadcastRequestWaitTime, RequestCreator requestCreator)	void dispose()
		void notifyResponseReceived(Response response)
		Map<String, Response> read(Data request, int rootBranchCount, int subBranchCount)
ClusterRootAnycastReader	ClusterRootAnycastReader(FreeClientPool clientPool, int multicastorPoolSize, long readerAssignWaitTime, long anycastRequestWaitTime, RequetCreator requestCreator)	void dispose()
		void notifyResponseReceived(Response response)
		Map<String, Response> read(Data request, int rootBranchCount, int subBranchCount)
ClusterRootUnicastReader	ClusterRootUnicastReader(FreeClientPool clientPool, int multicastorPoolSize, long readerAssignWaitTime, long anycastRequestWaitTime, RequetCreator requestCreator)	void dispose()
		void notifyResponseReceived(Response response)
		Map<String, Response> randomRead(Data request, int rootBranchCount, int subBranchCount)
		Map<String, Response> randomRead(Data request, int rootBranchCount, int subBranchCount, String childKey)
ClusterChildBroadcastNotifier	ClusterChildBroadcastNotifier(String localIPKey, FreeClientPool clientPool, int multicastorPoolSize, long notifierAssignWaitTime, NotificationCreator creator)	void dispose()
		void notify(Notification notification, int subBranchCount)
ClusterChildBroadcastReader	ClusterChildBroadcastReader(String localIPKey, FreeClientPool clientPool, int multicastorPoolSize, long readerAssignWaitTime, RequestCreator requestCreator)	void dispose()
		void read(Request request, int branchCount)

Table 7.2 The constructors and methods of multicasting APIs of GreatFree

According to Table 7.2, it is known that the methods of those APIs are simple and the number of those methods is small. Meanwhile, root APIs have identical their parameters for their constructors and so do children APIs. Additionally, the methods of root APIs are almost the same besides the unicasting one. You might ask why there are only two children APIs whereas the number of root ones is larger, i.e., six. You must notice that only broadcasting exists in the children APIs. That is easy to explain. When anycasting or unicasting data is received by a child node, it does not need to forward the data to its children to keep the multicasting since data received by the

node indicates that the goal of anycasting or unicasting is accomplished. Hence, no any children need to have anycasting and unicasting APIs.

Constructor	Explanation
ClusterRootBroadcastNotifier(FreeClientPool clientPool, int multicastorPoolSize, long notifierAssignWaitTime, NotificationCreator creator)	The three constructors serve for root APIs in the manner of broadcasting, anycasting and unicasting, respectively. They have the identical parameters as follows. • FreeClientPool clientPool: The tool that manages the instances of TCP client • int multicastorPoolSize: It needs to consume the resources of multicastor. So a built-in pool is designed for that. The size of the pool is specified here • long notifierAssignWaitTime: When the broadcast is busy, the built-in pool must not have enough resources to assign. If so, it is necessary to wait for some time • NotificationCreator creator: The creator that generates the instance of the notification
ClusterRootAnycastNotifier(FreeClientPool clientPool, int multicastorPoolSize, long notifierAssignWaitTime, NotificationCreator creator)	
ClusterRootUnicastNotifier(FreeClientPool clientPool, int multicastorPoolSize, long notifierAssignWaitTime, NotificationCreator creator)	
ClusterRootBroadcastReader(FreeClientPool clientPool, int multicastorPoolSize, long readerAssignWaitTime, long broadcastRequestWaitTime, RequestCreator creator)	The three constructors belong to the root APIs for request broadcasting. Its parameters are explained as follows. • FreeClientPool: The TCP client pool used for managing the resources of TCP clients • int multicastorPoolSize: The size of the built-in multicastor resource management pool • long readerAssignWaitTime: It is possible that the multicasting is busy such that no sufficient resources are available to create new multicastors. If so, it is necessary to wait for some time • long broadcastRequestWaitTime: When the request is broadcast to all of the children, it is necessary to set the time for waiting. It is possible that one child is failed or one child spends much time. If so, it is not reasonable to wait for ever. In the cases of anycasting and unicasting, the parameters are turned into anycastRequestWaitTime and unicastRequestWaitTime, respectively. The same as broadcastRequestWaitTime, both of them are also the values of timeout for the root • RequestCreator creator: The class creates instances of requests for the root
ClusterRootAnycastReader(FreeClientPool clientPool, int multicastorPoolSize, long readerAssignWaitTime, long anycastRequestWaitTime, RequetCreator creator)	
ClusterRootUnicastReader(FreeClientPool clientPool, int multicastorPoolSize, long readerAssignWaitTime, long unicastRequestWaitTime, RequetCreator creator)	
ClusterChildBroadcastNotifier(String localIPKey, FreeClientPool clientPool, int multicastorPoolSize, long notifierAssignWaitTime, NotificationCreator creator)	The constructors of children APIs have five parameters, four of which are identical to the corresponding ones in the root APIs. The one, localIPKey, is a new one added to them. • String localIPKey: The key of the local IP, which is unique to represent the local IP. Before the build-in multicastor needs to send messages, it is necessary to check whether the destination is the local one since it is unreasonable. The parameter is used to avoid the problem • FreeClientPool clientPool: The tool that manages the instances of TCP client • int multicastorPoolSize: It needs to consume the resources of multicastor. So a built-in pool is designed for that. The size of the pool is specified here • long notifierAssignWaitTime: When the broadcast is busy, the built-in pool must not have enough resources to assign. If so, it is necessary to wait for some time. For the case of reader, the corresponding parameter is readerAssignWaitTime • NotificationCreator creator: The creator that generates the instance of the notification. For the reader, the corresponding parameter is the "creator" in the type of RequestCreator
ClusterChildBroadcastReader(String localIPKey, FreeClientPool clientPool, int multicastorPoolSize, long readerAssignWaitTime, RequestCreator creator)	

Table 7.3 The explanations to the constructors of multicasting APIs

API	Method	Explanation
ClusterRootBroadcastNotifier	void dispose()	When the notification broadcasting is done, collect all of the resources by calling the method
	void notify(Data data, int rootBranchCount, int subBranchCount)	This is the method that achieves the goal of the notification broadcasting. It parameters are explained as follows. As a notifier, there are no return values for the method. • Data data: The data to be broadcast; it is defined in the form of Object • int rootBranchCount: Data is broadcast through a hierarchical topology. Usually, a tree is employed to do that. Since the root node is special, for example, having wider bandwidth, it is reasonable to allow

		developers to specify the branch count • int subBranchCount: The parameter specifies the any other branch count except the root one. In this case of the book, the value is identical to each of others. Such a configuration assumes that each child is identical in terms of their computing capacity and bandwidth. If the assumption is not valid, the sub branch count among children might be different
ClusterRootAnycastNotifier	void dispose()	When the notification anycasting is done, the method is invoked to collect all of the resources for the anycasting
	void notify(Data data, int rootBranchCount, int subBranchCount)	The form of the method is identical to the corresponding one of ClusterRootBroadcastNotifier
ClusterRootUnicastNotifier	void dispose()	Disposing the unicasting notifier when the unicasting is done.
	void randomNotify(Data data, int rootBranchCount, int subBranchCount)	The method aims to send notifications to one of the children in the cluster and the child is selected randomly. Its parameters are identical to the corresponding ones of ClusterRootBroadcastNotifier and ClusterRootAnycastNotifier
	void notify(Data data, int rootBranchCount, int subBranchCount, String childKey)	Different from the method to send notifications randomly, the method aims to send notifications to a specified child. So besides those parameters as others, it has one additional parameter, childKey in String, which represents the child in the cluster. Through it, the child can be located such that notifications can be sent to it
ClusterRootBroadcastReader	void dispose()	The method is invoked when the request broadcasting is done to collect the resources
	void notifyResponseReceived(Response response)	For a request broadcasting, each child is required to send one response to the root after receiving a request. When the root receives the response, it needs to invoke the method of the request broadcasting reader. Until the number of responses reaches to that of all of the children or the timeout is exceeded, the procedure is terminated. The method has one parameter, response, in the type of Response only
	Map<String, Response> read(Data request, int rootBranchCount, int subBranchCount)	This is the primary method of the request broadcasting. Its parameters are the same as those of multicasting notifiers. However, as a requesting broadcasting, it needs to obtain responses from all of the children in the cluster. So the method has a return value in the form of Map<String, Response>
ClusterRootAnycastReader	void dispose()	When the request anycasting is done, the method is used to collect all of the related resources
	void notifyResponseReceived(Response response)	The form of the method is exactly the same as the corresponding one of ClusterRootBroadcastReader. The difference is that the procedure of the request anycasting is terminated when one response is received at least
	Map<String, Response> read(Data request, int rootBranchCount, int subBranchCount)	The form of the method is also exactly the same as the corresponding one of ClusterRootBroadcastReader
ClusterRootUnicastReader	void dispose()	The method disposes the related resources after the request unicasting is done
	void notifyResponseReceived(Response response)	The form of the method is identical to that of the corresponding ones of ClusterRootBroadcastReader as well as ClusterRootAnycastReader. The difference is that only one response at most is received for a request unicasting. So the method is invoked at most once. If it is not invoked, it indicates that the system gets faults in the procedure
	Map<String, Response> randomRead(Data request, int rootBranchCount, int subBranchCount)	The form of the method is exactly the same as the corresponding ones of ClusterRootBroadcastReader and ClusterRootAnycastReader. However, the size of its return value is one at most

	Map<String, Response> randomRead(Data request, int rootBranchCount, int subBranchCount, String childKey)	The method has one additional parameter, childKey, which represents the child in the cluster the root attempts to send the request to. Other parameters are similar to those of its counterparts
ClusterChildBroadcastNotifier	void dispose()	The method is invoked such that the relevant resources can be collected
	void notify(Notification notification, int subBranchCount)	The method forwards notifications to its immediate children in the hierarchical structure. The number of the children is the value of subBranchCount
ClusterChildBroadcastReader	void dispose()	The method aims to collect the resources related to the procedure of broadcasting request
	void read(Request request, int branchCount)	The method forwards requests to its immediate children in the hierarchical structure. Although it is named as "read", it does not mean that child expects any responses from its children since all of the responses are sent back to the root

Table 7.4 The methods of multicasting APIs

Table 7.3 and Table 7.4 elaborate the constructors and their parameters of GreatFree multicasting APIs, respectively. I do not think they are so complicated that developers must feel afraid. You can think about the issue again. Through those APIs, developers can program a program that is able to send data to an infinite numbers of distributed nodes over the Internet. Compared with the ambitious goal, the APIs look more simple, right?

1.4 Programming a Cluster with Multicasting APIs

Different from previous chapters, like the chatting applications based on C/S or P2P, we do not present any particular applications in the chapter. Even though no applications, the content is still rich. We will focus on programming a cluster with multicasting APIs. Through the code we present, you can learn the skills to transmit data with a large-scale distributed system in order to form a cluster. Thereafter, you will be able to establish more complicated applications with the support of unlimited computing resources from the cluster.

1.4.1 The Relationship Between Clustering and Multicasting

The chapter talks about clustering programming. However, until now, all of APIs we have discussed are only related to multicasting. Actually, the technology of multicasting is the substrate of clustering. In most time, they can even be presented alternatively. From my point of view, an experienced programmer usually does not care about that if he knows how to solve his practical problems. When you program clusters with APIs in this chapter, you have to employ many multicasting APIs. However, after all of computers work together with the support of multicasting, we call the system, a cluster, rather than a single client, a single server or a single peer. It is unreasonable to call a computing system, multicasting. That is just a norm. If we still expect to differ them, I think clustering is more related to computing whereas multicasting is focused on communicating. For example, a cluster can behave in the way of a client, a server or a peer, like what we talk about previously. Well, multicasting is always about data transmission.

1.4.2 The Distributed Node In a Cluster

Before we touch the exact code, let us solve another problem. The question is which types of distributed components would you like to choose for a cluster. We have learned three types of them in the previous chapters, including a client, a server or a peer. Which ones are suitable to the cluster? The choice results in the different distributed models, i.e., C/S or P2P, within a cluster. Actually, I have ever used some frameworks for clusters that are based on C/S, such as Solr [2]. When data needs to be moved from one node to other ones which are organized in a hierarchical structure, the technique of polling, which we discuss in Chapter 5, must be utilized. Lower level nodes must poll their immediately higher level ones to obtain data. On other words, high level nodes never distribute data actively to their immediately lower level nodes. It looks weird. If a node is able to send polling requests to its parent, why cannot it send data to its children? If a node is able to receive requests from its children, why cannot it receive data from its parent? In such a system, such a weird node is the distributed component, which has a client to send polling requests to its parent as well as a server to receive such requests from its children. In fact, the component is a peer indeed. But if so, polling is not necessary. But in reality, such a cluster is constructed by one-to-one polling rather than multicasting.

I guess that there are two primary reasons for that. First of all, such a system, like Solr, employs Tomcat [3] as the server. Tomcat is a typical dedicated Web server. So it does not consider sufficient issues that occur in the real world distributed computing environment. Instead, it focuses on working as a standalone node to receive HTTP [4] requests and respond to clients with HTTP responses only. At least, multicasting is not the important mechanism when it was designed. In addition, a peer-based system, as we elaborate in Chapter 6, must have a registry server. When deploying Tomcat-based applications, neither is such a registry server required for developers to implement, nor are there built-in modules for that either. So, it is impossible to establish a peer-based system.

Therefore, in the system we will program, the distributed component, Peer, is the only possible choice for clusters. We already have the registry server and no anything needs to be updated on that. If you forget, just review Section 2 of Chapter 6. Moreover, the multicasting APIs are also available, as presented in Section 1 of this chapter. For that, we are ready for programming right now.

2. Programming the Root

As presented previously, we will not program one specific application in the chapter. However, what we will do is to employ all of the multicasting APIs in one process such that we can learn how they work and how we can apply them to program clusters and large-scale distributed systems.

2.1 SP: Starting Up the Root

According to our convention, we need to find out whether there are patterns in the code. In brief, APIs and patterns are the tools you can rely on to program distributed systems with GreatFree.

List 7.1 shows the code to start the root of a cluster.

```
1   package com.greatfree.cluster.root;
2
3   import java.io.IOException;
4   import java.util.Scanner;
5
6   import com.greatfree.chat.MenuOptions;
7   import com.greatfree.exceptions.RemoteReadException;
8   import com.greatfree.testing.client.ClientMenu;
9   import com.greatfree.testing.data.ServerConfig;
10  import com.greatfree.util.ServerStatus;
11  import com.greatfree.util.TerminateSignal;
12
13  /*
14   * The class starts up the cluster and tests it whether data can be multicast from it. 05/08/2017, Bing Li
15   */
16
17  // Created: 05/08/2017, Bing Li
18  public class StartRoot
19  {
20      public static void main(String[] args)
21      {
22          // Initialize the option which represents a user's intents of operations. 09/21/2014, Bing Li
23          int option = MenuOptions.NO_OPTION;
24
25          // Initialize a command input console for users to interact with the system. 09/21/2014, Bing Li
26          Scanner in = new Scanner(System.in);
27          String optionStr;
28
29          System.out.println("Cluster root starting up ...");
30
31          try
32          {
33              // Start up the cluster root. 06/11/2017, Bing Li
34              ClusterRoot.CLUSTER().start();
35          }
36          catch (ClassNotFoundException | IOException | RemoteReadException | InstantiationException |
37              IllegalAccessException | InterruptedException e)
38          {
39              e.printStackTrace();
40          }
41          System.out.println("Cluster root started ...");
42
43          // Keep the loop running to interact with users until an end option is selected. 09/21/2014, Bing Li
44          while (option != MenuOptions.QUIT)
45          {
46              // Display the menu to users. 09/21/2014, Bing Li
47              ClusterRootUI.CLUSTER().printMenu();
48              // Input a string that represents users' intents. 09/21/2014, Bing Li
49              optionStr = in.nextLine();
50              // Convert the input string to integer. 09/21/2014, Bing Li
51              option = Integer.parseInt(optionStr);
52              System.out.println("Your choice: " + option);
53              try
54              {
55                  // Send the option to the cluster. 09/21/2014, Bing Li
56                  ClusterRootUI.CLUSTER().send(option);
57              }
58              catch (ClassNotFoundException | InstantiationException | IllegalAccessException |
59                  RemoteReadException | IOException | InterruptedException | NumberFormatException e)
60              {
61                  option = MenuOptions.NO_OPTION;
62                  System.out.println(ClientMenu.WRONG_OPTION);
63              }
64          }
```

```
65
66          // Set the flag of the server as down. 05/15/2017, Bing Li
67          ServerStatus.FREE().setShutdown();
68          TerminateSignal.SIGNAL().setTerminated();
69          try
70          {
71              ClusterRoot.CLUSTER().stop();
72          }
73          catch (IOException | InterruptedException | ClassNotFoundException | RemoteReadException e)
74          {
75              e.printStackTrace();
76          }
77
78          // After the server is started, the loop check whether the flag of terminating is set. If the terminating flag
79          // is true, the process is ended. Otherwise, the process keeps running. 08/22/2014, Bing Li
80          while (!TerminateSignal.SIGNAL().isTerminated())
81          {
82              try
83              {
84                  // If the terminating flag is false, it is required to sleep for some time. Otherwise, it might cause the
85                  // high CPU usage. 08/22/2014, Bing Li
86                  Thread.sleep(ServerConfig.TERMINATE_SLEEP);
87              }
88              catch (InterruptedException e)
89              {
90                  e.printStackTrace();
91              }
92          }
93          in.close();
94      }
95  }
```

List 7.1 The code of StartRoot.java

Before presenting code in detail, you should be aware of the fact that the root is implemented over the API of Peer. If so, the starting point of the code and the one of List 6.42, which is also a peer, must have the same structure. That is exactly true! Removing those lines (Line 25~30) about specific applications from List 6.42, both of them are identical in terms of their patterns of SP. For that, the pattern can be summarized in Table 7.5.

Pattern: SP (Starting Point)		
Class	StartRoot	
Singleton	Yes	
Method	void main(String[] args)	
Enclosed Pattern	MS (Main Server)	ClusterRoot
Employed API	TerminalSignal	

Table 7.5 The pattern of SP for the cluster root

Line 31~40 starts up the root, which is mainly implemented by the API of Peer.

Line 44~64 displays the UI to the operator of the root.

Line 67~76 shuts down the root.

2.2 CR: Programming the Client Side of the Cluster Root

We now need to program the root of the cluster. As a cluster, the root is responsible for organizing the multicasting topology and sending initial messages into the hierarchical structure. For that, the code of the root should be brand new. You must see many multicasting APIs designed for the root.

You can refer Table 7.1~7.4. List 7.2 shows the code of ClusterRoot.java. Fortunately, you do not need to scare about the code if a pattern can be summarized from the code.

```
1   package com.greatfree.cluster.root;
2
3   import java.io.IOException;
4   import java.util.Map;
5
6   import com.greatfree.chat.ChatConfig;
7   import com.greatfree.cluster.ClusterConfig;
8   import com.greatfree.cluster.HelloWorld;
9   import com.greatfree.cluster.message.HelloWorldAnycastNotification;
10  import com.greatfree.cluster.message.HelloWorldAnycastRequest;
11  import com.greatfree.cluster.message.HelloWorldAnycastResponse;
12  import com.greatfree.cluster.message.HelloWorldBroadcastNotification;
13  import com.greatfree.cluster.message.HelloWorldBroadcastRequest;
14  import com.greatfree.cluster.message.HelloWorldBroadcastResponse;
15  import com.greatfree.cluster.message.HelloWorldUnicastNotification;
16  import com.greatfree.cluster.message.HelloWorldUnicastRequest;
17  import com.greatfree.cluster.message.HelloWorldUnicastResponse;
18  import com.greatfree.cluster.message.RootIPAddressBroadcastNotification;
19  import com.greatfree.cluster.message.ShutdownChildrenBroadcastNotification;
20  import com.greatfree.cluster.message.root.HelloWorldAnycastNotificationCreator;
21  import com.greatfree.cluster.message.root.HelloWorldAnycastRequestCreator;
22  import com.greatfree.cluster.message.root.HelloWorldBroadcastNotificationCreator;
23  import com.greatfree.cluster.message.root.HelloWorldBroadcastRequestCreator;
24  import com.greatfree.cluster.message.root.HelloWorldUnicastNotificationCreator;
25  import com.greatfree.cluster.message.root.HelloWorldUnicastRequestCreator;
26  import com.greatfree.cluster.message.root.RootIPAddressBroadcastNotificationCreator;
27  import com.greatfree.cluster.message.root.ShutdownChildrenBroadcastNotificationCreator;
28  import com.greatfree.clustering.root.ClusterRootAnycastNotifier;
29  import com.greatfree.clustering.root.ClusterRootAnycastReader;
30  import com.greatfree.clustering.root.ClusterRootBroadcastNotifier;
31  import com.greatfree.clustering.root.ClusterRootBroadcastReader;
32  import com.greatfree.clustering.root.ClusterRootUnicastNotifier;
33  import com.greatfree.clustering.root.ClusterRootUnicastReader;
34  import com.greatfree.exceptions.RemoteReadException;
35  import com.greatfree.message.ClusterIPRequest;
36  import com.greatfree.message.ClusterIPResponse;
37  import com.greatfree.server.Peer;
38  import com.greatfree.testing.data.ServerConfig;
39  import com.greatfree.util.IPAddress;
40  import com.greatfree.util.UtilConfig;
41
42  /*
43   * The class, CSCluster, is defined as a system enclosing multiple or infinite number of distributed instead of
44   * just one, like the CSServer or the Peer. All of the composed distributed node forms a single node from a
45   * developer's point of view. 05/04/2017, Bing Li
46   */
47
48  // Created: 05/04/2017, Bing Li
49  public class ClusterRoot
50  {
51      // The main node of the cluster. 05/10/2017, Bing Li
52      private Peer<RootDispatcher> peer;
53
54      // Broadcast the root IP address to each child in the cluster. It is required to perform broadcast
55      // request/response. So this is a system level broadcast notifier. 05/10/2017, Bing Li
56      private ClusterRootBroadcastNotifier<IPAddress, RootIPAddressBroadcastNotification,
57          RootIPAddressBroadcastNotificationCreator> rootIPBroadcastNotifier;
58      // One broadcast notifier. 05/10/2017, Bing Li
59      private ClusterRootBroadcastNotifier<HelloWorld, HelloWorldBroadcastNotification,
60          HelloWorldBroadcastNotificationCreator> helloWorldBroadcastNotifier;
61      // One anycast notifier. 05/19/2017, Bing Li
62      private ClusterRootAnycastNotifier<HelloWorld, HelloWorldAnycastNotification,
63          HelloWorldAnycastNotificationCreator> helloWorldAnycastNotifier;
64      // One unicast notifier. 05/19/2017, Bing Li
65      private ClusterRootUnicastNotifier<HelloWorld, HelloWorldUnicastNotification,
66          HelloWorldUnicastNotificationCreator> helloWorldUnicastNotifier;
67      // One broadcast reader. 05/20/2017, Bing Li
68      private ClusterRootBroadcastReader<HelloWorld, HelloWorldBroadcastRequest,
69          HelloWorldBroadcastResponse, HelloWorldBroadcastRequestCreator> helloWorldBroadcastReader;
```

```
70      // One anycast reader. 05/21/2017, Bing Li
71      private ClusterRootAnycastReader<HelloWorld, HelloWorldAnycastRequest,
72          HelloWorldAnycastResponse, HelloWorldAnycastRequestCreator> helloWorldAnycastReader;
73      // One unicast reader. 05/21/2017, Bing Li
74      private ClusterRootUnicastReader<HelloWorld, HelloWorldUnicastRequest, HelloWorldUnicastResponse,
75          HelloWorldUnicastRequestCreator> helloWorldUnicastReader;
76      // One broadcast notifier to send shutdown children broadcast notification. 05/19/2017, Bing Li
77      private ClusterRootBroadcastNotifier<HelloWorld, ShutdownChildrenBroadcastNotification,
78          ShutdownChildrenBroadcastNotificationCreator> shutdownChildrenBroadcastNotifier;
79
80      private ClusterRoot()
81      {
82      }
83
84      private static ClusterRoot instance = new ClusterRoot();
85
86      public static ClusterRoot CLUSTER()
87      {
88          if (instance == null)
89          {
90              instance = new ClusterRoot();
91              return instance;
92          }
93          else
94          {
95              return instance;
96          }
97      }
98
99
100     /*
101      * Stop the CSCluster. 05/08/2017, Bing Li
102      */
103     public void stop() throws IOException, InterruptedException, ClassNotFoundException,
104         RemoteReadException
105     {
106         // Stop the peer. 04/30/2017, Bing Li
107         this.peer.stop();
108
109         // Dispose the root IP address broadcast notifier. 05/10/2017, Bing Li
110         this.rootIPBroadcastNotifier.dispose();
111         // Dispose the broadcast notifier. 05/10/2017, Bing Li
112         this.helloWorldBroadcastNotifier.dispose();
113         // Dispose the anycast notifier. 05/10/2017, Bing Li
114         this.helloWorldAnycastNotifier.dispose();
115         // Dispose the unicast notifier. 05/10/2017, Bing Li
116         this.helloWorldUnicastNotifier.dispose();
117         // Dispose the broadcast reader. 05/20/2017, Bing Li
118         this.helloWorldBroadcastReader.dispose();
119         // Dispose the anycast reader. 05/21/2017, Bing Li
120         this.helloWorldAnycastReader.dispose();
121         // Dispose the unicast reader. 05/21/2017, Bing Li
122         this.helloWorldUnicastReader.dispose();
123         // Dispose the children shutdown broadcast notifier. 05/19/2017, Bing Li
124         this.shutdownChildrenBroadcastNotifier.dispose();
125     }
126
127     /*
128      * Start the CSCluster. 05/08/2017, Bing Li
129      */
130     public void start() throws ClassNotFoundException, RemoteReadException, IOException,
131         InstantiationException, IllegalAccessException, InterruptedException
132     {
133         // Initialize the peer. 06/13/2017, Bing Li
134         this.peer = new Peer.PeerBuilder<RootDispatcher>()
135             .peerPort(ChatConfig.CHAT_SERVER_PORT)
136             .peerName(ClusterConfig.CLUSTER_ROOT_NAME)
137             .sysRegistryServerIP(ChatConfig.CHAT_REGISTRY_ADDRESS)
138             .sysRegistryServerPort(UtilConfig.PEER_REGISTRY_PORT)
139             .listenerCount(ServerConfig.LISTENING_THREAD_COUNT)
140             .listenerThreadPoolSize(ServerConfig.SHARED_THREAD_POOL_SIZE)
141             .listenerThreadKeepAliveTime
142                 (ServerConfig.SHARED_THREAD_POOL_KEEP_ALIVE_TIME)
143             .dispatcher(new RootDispatcher(ChatConfig.DISPATCHER_THREAD_POOL_SIZE,
144                 ChatConfig.DISPATCHER_THREAD_POOL_KEEP_ALIVE_TIME,
145                 ChatConfig.SCHEDULER_THREAD_POOL_SIZE,
146                 ChatConfig.SCHEDULER_THREAD_POOL_KEEP_ALIVE_TIME))
```

```
147             .clientPoolSize(ChatConfig.CLIENT_POOL_SIZE)
148             .readerClientSize(ChatConfig.READER_CLIENT_SIZE)
149             .syncEventerIdleCheckDelay(ChatConfig.SYNC_EVENTER_IDLE_CHECK_DELAY)
150             .syncEventerIdleCheckPeriod(ChatConfig.SYNC_EVENTER_IDLE_CHECK_PERIOD)
151             .syncEventerMaxIdleTime(ChatConfig.SYNC_EVENTER_MAX_IDLE_TIME)
152             .asyncEventQueueSize(ChatConfig.ASYNC_EVENT_QUEUE_SIZE)
153             .asyncEventerSize(ChatConfig.ASYNC_EVENTER_SIZE)
154             .asyncEventingWaitTime(ChatConfig.ASYNC_EVENTING_WAIT_TIME)
155             .asyncEventerWaitTime(ChatConfig.ASYNC_EVENTER_WAIT_TIME)
156             .asyncEventerWaitRound(ChatConfig.ASYNC_EVENTER_WAIT_ROUND)
157             .asyncEventIdleCheckDelay(ChatConfig.ASYNC_EVENT_IDLE_CHECK_DELAY)
158             .asyncEventIdleCheckPeriod(ChatConfig.ASYNC_EVENT_IDLE_CHECK_PERIOD)
159             .clientThreadPoolSize(ChatConfig.CLIENT_THREAD_POOL_SIZE)
160             .clientThreadKeepAliveTime(ChatConfig.CLIENT_THREAD_KEEP_ALIVE_TIME)
161             .schedulerPoolSize(ChatConfig.SCHEDULER_THREAD_POOL_SIZE)
162             .schedulerKeepAliveTime(ChatConfig.SCHEDULER_THREAD_POOL_KEEP_ALIVE_TIME)
163             .build();
164
165         // Start up the peer. 06/14/2017, Bing Li
166         this.peer.start();
167
168         // Retrieve all of the registered IP addresses of the distributed nodes in the cluster from the
169         // registry server. 05/08/2017, Bing Li
170         ClusterIPResponse ipResponse = (ClusterIPResponse)this.peer.read
171             (ChatConfig.CHAT_REGISTRY_ADDRESS,  UtilConfig.PEER_REGISTRY_PORT,
172             new ClusterIPRequest());
173
174         // Add the IP addresses to the client pool. 05/08/2017, Bing Li
175         for (IPAddress ip : ipResponse.getIPs().values())
176         {
177             this.peer.getClientPool().addIP(ip.getKey(), ip.getIP(), ip.getPort());
178         }
179
180         // Initialize the cluster root IP address broadcast notifier. 05/08/2017, Bing Li
181         this.rootIPBroadcastNotifier = new ClusterRootBroadcastNotifier<IPAddress,
182             RootIPAddressBroadcastNotification, RootIPAddressBroadcastNotificationCreator>
183             (this.peer.getClientPool(), ClusterConfig.MULTICASTOR_POOL_SIZE,
184             ClusterConfig.RESOURCE_WAIT_TIME, new RootIPAddressBroadcastNotificationCreator());
185
186         // Broadcast the root IP address to each child in the cluster. 05/20/2017, Bing Li
187         this.rootIPBroadcastNotifier.notifiy(new IPAddress(this.peer.getID(), this.peer.getPeerIP(),
188             this.peer.getPort()), ClusterConfig.ROOT_BRANCH_COUNT,
189             ClusterConfig.SUB_BRANCH_COUNT);
190
191         // Initialize the cluster root broadcast notifier. 05/08/2017, Bing Li
192         this.helloWorldBroadcastNotifier = new ClusterRootBroadcastNotifier<HelloWorld,
193             HelloWorldBroadcastNotification, HelloWorldBroadcastNotificationCreator>
194             (this.peer.getClientPool(), ClusterConfig.MULTICASTOR_POOL_SIZE,
195             ClusterConfig.RESOURCE_WAIT_TIME, new HelloWorldBroadcastNotificationCreator());
196
197         // Initialize the cluster root anycast notifier. 05/19/2017, Bing Li
198         this.helloWorldAnycastNotifier = new ClusterRootAnycastNotifier<HelloWorld,
199             HelloWorldAnycastNotification, HelloWorldAnycastNotificationCreator>(this.peer.getClientPool(),
200             ClusterConfig.MULTICASTOR_POOL_SIZE, ClusterConfig.RESOURCE_WAIT_TIME, new
201             HelloWorldAnycastNotificationCreator());
202
203         // Initialize the cluster root unicast notifier. 05/19/2017, Bing Li
204         this.helloWorldUnicastNotifier = new ClusterRootUnicastNotifier<HelloWorld,
205             HelloWorldUnicastNotification, HelloWorldUnicastNotificationCreator>(this.peer.getClientPool(),
206             ClusterConfig.MULTICASTOR_POOL_SIZE, ClusterConfig.RESOURCE_WAIT_TIME,
207             new HelloWorldUnicastNotificationCreator());
208
209         // Initialize the cluster root broadcast reader. 05/20/2017, Bing Li
210         this.helloWorldBroadcastReader = new ClusterRootBroadcastReader<HelloWorld,
211             HelloWorldBroadcastRequest, HelloWorldBroadcastResponse,
212             HelloWorldBroadcastRequestCreator>(this.peer.getClientPool(),
213             ClusterConfig.MULTICASTOR_POOL_SIZE, ClusterConfig.RESOURCE_WAIT_TIME,
214             ClusterConfig.BROADCAST_REQUEST_WAIT_TIME,
215             new HelloWorldBroadcastRequestCreator());
216
217         // Initialize the cluster root anycast reader. 05/20/2017, Bing Li
218         this.helloWorldAnycastReader = new ClusterRootAnycastReader<HelloWorld,
219             HelloWorldAnycastRequest, HelloWorldAnycastResponse, HelloWorldAnycastRequestCreator>
220             (this.peer.getClientPool(), ClusterConfig.MULTICASTOR_POOL_SIZE,
221             ClusterConfig.RESOURCE_WAIT_TIME,
222             ClusterConfig.BROADCAST_REQUEST_WAIT_TIME,
223             new HelloWorldAnycastRequestCreator());
```

```
224
225         // Initialize the cluster root unicast reader. 05/20/2017, Bing Li
226         this.helloWorldUnicastReader = new ClusterRootUnicastReader<HelloWorld,
227             HelloWorldUnicastRequest, HelloWorldUnicastResponse,
228             HelloWorldUnicastRequestCreator>(this.peer.getClientPool(),
229             ClusterConfig.MULTICASTOR_POOL_SIZE, ClusterConfig.RESOURCE_WAIT_TIME,
230             ClusterConfig.BROADCAST_REQUEST_WAIT_TIME,
231             new HelloWorldUnicastRequestCreator());
232
233         // Initialize the cluster root shutdown children notifier. 05/19/2017, Bing Li
234         this.shutdownChildrenBroadcastNotifier = new ClusterRootBroadcastNotifier<HelloWorld,
235             ShutdownChildrenBroadcastNotification, ShutdownChildrenBroadcastNotificationCreator>
236             (this.peer.getClientPool(), ClusterConfig.MULTICASTOR_POOL_SIZE,
237             ClusterConfig.RESOURCE_WAIT_TIME,
238             new ShutdownChildrenBroadcastNotificationCreator());
239     }
240
241     /*
242      * Broadcast the notification to the distributed nodes in the cluster. 05/15/2017, Bing Li
243      */
244     public void broadcastNotify(HelloWorld hw, int rootBranchCount, int subBranchCount) throws
245         InstantiationException, IllegalAccessException, IOException, InterruptedException
246     {
247         this.helloWorldBroadcastNotifier.notifiy(hw, rootBranchCount, subBranchCount);
248     }
249
250     /*
251      * Anycast the notification to the distributed nodes in the cluster. 05/15/2017, Bing Li
252      */
253     public void anycastNotify(HelloWorld hw, int rootBranchCount, int subBranchCount) throws
254         InstantiationException, IllegalAccessException, IOException, InterruptedException
255     {
256         this.helloWorldAnycastNotifier.notifiy(hw, rootBranchCount, subBranchCount);
257     }
258
259     /*
260      * Unicast the notification to the distributed nodes in the cluster. 05/15/2017, Bing Li
261      */
262     public void unicastNotify(HelloWorld hw, int rootBranchCount, int subBranchCount) throws
263         InstantiationException, IllegalAccessException, IOException, InterruptedException
264     {
265         this.helloWorldUnicastNotifier.notifiy(hw, rootBranchCount, subBranchCount);
266     }
267
268     /*
269      * Broadcast the shutdown notification to the distributed nodes in the cluster. 05/15/2017, Bing Li
270      */
271     public void broadcastShutdownNotify(int rootBranchCount, int subBranchCount) throws
272         InstantiationException, IllegalAccessException, IOException, InterruptedException
273     {
274         this.shutdownChildrenBroadcastNotifier.notifiy(null, rootBranchCount, subBranchCount);
275     }
276
277     /*
278      * Broadcast the request to the distributed nodes in the cluster. 05/20/2017, Bing Li
279      */
280     public Map<String, HelloWorldBroadcastResponse> broadcastRead(HelloWorld hw, int rootBranchCount,
281         int subBranchCount) throws InstantiationException, IllegalAccessException, InterruptedException,
282         IOException
283     {
284         return this.helloWorldBroadcastReader.read(hw, rootBranchCount, subBranchCount);
285     }
286
287     /*
288      * Collect responses from all of the children in the cluster until timeout. 05/21/2017, Bing Li
289      */
290     public void broadcastResponseReceived(HelloWorldBroadcastResponse response)
291     {
292         this.helloWorldBroadcastReader.notifyResponseReceived(response);
293     }
294
295     /*
296      * Anycast the shutdown notification to the distributed nodes in the cluster. 05/15/2017, Bing Li
297      */
298     public Map<String, HelloWorldAnycastResponse> anycastRead(HelloWorld hw, int rootBranchCount,
299         int subBranchCount) throws InstantiationException, IllegalAccessException, InterruptedException,
300         IOException
```

```
301         {
302             return this.helloWorldAnycastReader.read(hw, rootBranchCount, subBranchCount);
303         }
304
305         /*
306          * Collect responses from any of the children in the cluster until time. 05/21/2017, Bing Li
307          */
308         public void anycastResponseReceived(HelloWorldAnycastResponse response)
309         {
310             this.helloWorldAnycastReader.notifyResponseReceived(response);
311         }
312
313         /*
314          * Unicast the shutdown notification to the distributed nodes in the cluster. 05/15/2017, Bing Li
315          */
316         public Map<String, HelloWorldUnicastResponse> unicastRead(HelloWorld hw, int rootBranchCount,
317             int subBranchCount) throws InstantiationException, IllegalAccessException, InterruptedException,
318             IOException
319         {
320             return this.helloWorldUnicastReader.read(hw, rootBranchCount, subBranchCount);
321         }
322
323         /*
324          * Collect responses from the particular one of the children in the cluster until time. 05/21/2017, Bing Li
325          */
326         public void unicastResponseReceived(HelloWorldUnicastResponse response)
327         {
328             this.helloWorldUnicastReader.notifyResponseReceived(response);
329         }
330     }
```

List 7.2 The code of ClusterRoot.java

Table 7.6 gives a summary about the pattern of Cluster Root (CR) for the cluster you are programming. According to the table, you know that the structure of the cluster root is also highly patterned. The cluster root is made up with one SD, Peer<RootDispatcher>, and two multicasting-related patterns, i.e., MN (Multicasting Notifier) and MR (Multicasting Reader). Both MN and MR correspond with those multicasting APIs. However, those patterns just initialize and invoke the multicasting APIs such that we do not care about them too much. The reason to indicate them is to explain the pattern of CR conveniently. Anyway, the code of cluster root is still highly patterned such that you can still keep on programming it with the simplified behaviors, CPR.

Line 52 declares an instance of Peer, peer, which receives messages from the children of the cluster in the case.

Line 56 declares an instance of ClusterRootBroadcastNotifer, rootIPBroadcastNotifier, which broadcast the root IP address to all of the children in the cluster such that any child is able to send responses to the root.

Line 59 declares an instance of ClusterRootBroadcastNotifier, helloWorldBroadcastNotifier, which is responsible for broadcasting notifications to each of children in the cluster. The data to be broadcast is the instance of the object, HelloWorld enclosed in the notification, HelloWorldBroadcastNotification.

Line 62 declares an instance of ClusterRootAnycastNotifier, helloWorldAnycastNotifier. It sends the object, HelloWorld enclosed in the notification of HelloWorldAnycastNotification, to any of the children in the cluster.

Pattern: CR (Cluster Root)		
Class	ClusterRoot	
Singleton	Yes	
Method	void start()	
	void stop()	
	void broadcastNotify(Serializable object, int rootBranchCount, int subBranchCount)	
	void anycastNotify(Serializable object, int rootBranchCount, int subBranchCount)	
	void unicastNotify(Serializable object, int rootBranchCount, int subBranchCount)	
	void broadcastShutdownNotify(int rootBranchCount, int subBranchCount)	
	Map<String, ClusterBroadcastResponse> broadcastRead(Serializable object, int rootBranchCount, int subBranchCount)	
	void broadcastResponseReceived(ClusterBroadcastResponse response)	
	Map<String, ClusterAnycastResponse> anycastRead(Serializable object, int rootBranchCount, int subBranchCount)	
	void anycastResponseReceived(ClusterBroadcastResponse response)	
	Map<String, ClusterAnycastResponse> unicastRead(Serializable object, int rootBranchCount, int subBranchCount)	
	void unicastResponseReceived(ClusterBroadcastResponse response)	
Enclosed Pattern	SD	Peer<RootDispatcher>
	MN (Multicasting Notifier)	ClusterRootBroadcastNotifier<IPAddress, RootIPAddressBroadcastNotification, RootIPAddressBroadcastNotificationCreator>
		ClusterRootBroadcastNotifier<HelloWorld, HelloWorldBroadcastNotification, HelloWorldBroadcastNotificationCreator>
		ClusterRootBroadcastNotifier<HelloWorld, ShutdownChildrenBroadcastNotification, ShutdownChildrenBroadcastNotificationCreator>
		ClusterRootAnycastNotifier<HelloWorld, HelloWorldAnycastNotification, HelloWorldAnycastNotificationCreator>
		ClusterRootUnicastNotifier<HelloWorld, HelloWorldUnicastNotification, HelloWorldUnicastNotificationCreator>
	MR (Multicasting Reader)	ClusterRootBroadcastReader<HelloWorld, HelloWorldBroadcastRequest, HelloWorldBroadcastResponse, HelloWorldBroadcastRequestCreator>
		ClusterRootAnycastReader<HelloWorld, HelloWorldAnycastRequest, HelloWorldAnycastResponse, HelloWorldAnycastRequestCreator>
		ClusterRootUnicastReader<HelloWorld, HelloWorldUnicastRequest, HelloWorldUnicastResponse, HelloWorldUnicastRequestCreator>
Employed API	Peer<Dispatcher extends ServerDispatcher>	
	ClusterRootBroadcastNotifier<Data extends Serializable, Notification extends MulticastMessage, NotificationCreator extends RootBroadcastNotificationCreatable<Notification, Data>>	
	ClusterRootAnycastNotifier<Data extends Serializable, Notification extends MulticastMessage, NotificationCreator extends RootBroadcastNotificationCreatable<Notification, Data>>	
	ClusterRootUnicastNotifier<Data extends Serializable, Notification extends MulticastMessage, NotificationCreator extends RootBroadcastNotificationCreatable<Notification, Data>>	
	ClusterRootBroadcastReader<Data extends Serializable, Request extends ClusterBroadcastRequest, Response extends ClusterBroadcastResponse, RequestCreator extends RootBroadcastRequestCreatable<Request, Data>>	
	ClusterRootAnycastReader<Data extends Serializable, Request extends ClusterBroadcastRequest, Response extends ClusterBroadcastResponse, RequestCreator extends RootBroadcastRequestCreatable<Request, Data>>	
	ClusterRootUnicastReader<Data extends Serializable, Request extends ClusterBroadcastRequest, Response extends ClusterBroadcastResponse, RequestCreator extends RootBroadcastRequestCreatable<Request, Data>>	

Table 7.6 The pattern of CR for the cluster root

Line 65 declares an instance of ClusterRootUnicastNotifier, helloWorldUnicastNotifier. It sends the object of HelloWorld to just a single child in the cluster. In this case, the child is selected randomly instead of specifying the one explicitly.

Line 68 declares an instance of ClusterRootBroadcastReader, helloWorldBroadcastReader. It sends a request, HelloWorldBroadcastRequest which encloses the object, HelloWorld, to each child in the cluster and then waits for responses from each of them so as to generate the final response.

Line 71 declares an instance of ClusterRootAnycastReader, helloWorldAnycastReader. It sends a request, HelloWorldAnycastRequest which encloses the object, HelloWorld, to any children in the cluster and then waits for responses from any of them so as to generate the final response.

Line 74 declares an instance of ClusterRootUnicastReader, helloWorldUnicastReader. It sends a request, HelloWorldUnicastRequest enclosing the object, HelloWorld, to a single child in the cluster and waits for it response to generate the final one.

Line 77 declares an instance of ClusterRootBroadcastNotifier, shutdownChildrenBroadcastNotifier, which sends an notification, ShutdownChildrenBroadcastNotification, to each child in the cluster so as to shut down them.

Line 80~97 defines the cluster root as a singleton.

Line 103~125 defines the method, stop(), to terminate the cluster root. Besides shutdown the peer in Line 1097 within the method, each of the instances of multicasting should be disposed in Line 110~124.

Line 130~235 defines the method, start(), to start up the cluster root.

Line 134~163 invokes the constructor of Peer to initialize the instance, peer. This code is exactly the same as Line 72~101 of ChatPeerSingleton in List 6.43.

Line 166 starts up the peer.

Line 170~178 is new. As the root of the cluster, it needs to organize those children into a hierarchical structure. To do that, it needs to get all of the IDs and IP addresses of those children.

Line 170~172 sends a request, ClusterIPRequest, to the registry server. Then, it gets the response, ClusterIPResponse, from the server. The response includes all of registered IDs and their IP addresses.

Line 175~178 presents a loop, in which the registered ID and IP addresses obtained from the registry servers are kept in the TCP client pool.

Line 181~184 initializes the instance, rootIPBroadcastNotifier, of ClusterRootBroadcastNotifier. Its constructor is explained in Table 7.3.

In Line 187~189, the instance, rootIPBroadcastNotifier, of ClusterRootBroadcastNotifier sends a broad notification, which encloses the IP address of the root, to all of the children in the cluster. It is required for a cluster since all of the children in the cluster needs to interact with the root when necessary. For

example, they must have the root IP address to send responses to broadcast requests to the root.

Line 192~195 initializes the instance, helloWorldBroadcastNotifer, of ClusterRootBroadcastNotifier. Its constructor is explained in Table 7.3.

Line 198~201 initializes the instance, helloWorldAnycastNotifier, of ClusterRootAnycastNotifier. Its constructor is explained in Table 7.3

Line 204~207 initializes the instance, helloWorldUnicastNotifier, of ClusterRootUnicastNotifier. Its constructor is explained in Table 7.3.

Line 210~215 initializes the instance, helloWorldBroadcastReader, of ClusterRootBroadcastReader. Its constructor is explained in Table 7.3.

Line 218~223 initializes the instance, helloWorldAnycastReader, of ClusterRootAnycastReader. Its constructor is explained in Table 7.3.

Line 226~231 initializes the instance, helloWorldUnicastReader, of ClusterRootUnicastReader. Its constructor is explained in Table 7.3.

Line 234~238 initializes the instance, shutdownChildrenBroadcastNotifier, of ClusterRootBroadcastNotifier. Its constructor is explained in Table 7.3.

Line 244~329 is also new to you. They perform multicasting within the cluster.

Line 244~248 defines the method, broadcastNotify(), to send one instance, hw, of HelloWorld to all of the children within the cluster.

Line 253~257 defines the method, anycastNotify(), to send one instance, hw, of HelloWorld to any of the children within the cluster.

Line 262~265 defines the method, unicastNotify(), to send one instance, hw, of HelloWorld to a single child which is chosen randomly within the cluster.

Line 271~275 defines the method, broadcastShutdownNotify(), to send the shutting down notification to all of the children within the cluster.

Line 280~285 defines the method, broadcastRead(), to send a request enclosing the instance, hw, of HelloWorld to all of the children within the cluster.

Line 290~293 defines the method, broadcastResponseReceived(), which is called when a response from a child is received.

Line 298~303 defines the method, anycastRead(), to send a request enclosing the instance, hw, of HelloWorld to any of the children within the cluster.

Line 308~311 defines the method, anycastResponseReceived(), which is called when a response from a child is received.

Line 316~321 defines the method, unicastRead(), to send a request enclosing the instance, hw, of HelloWorld to one unique child within the cluster.

Line 326~329 defines the method, unicastResponseReceived(), which is called when a response from the unique child is received.

Until now the client side of the cluster root is introduced completely. Although there are other code, only the one in the section is crucial. According to List 7.2, you must find that it is not so difficult to program the cluster root. To a great extent, it is actually like a classic client as we have talked in Section 3 of Chapter 5. You can compare List 7.2 with the code in List 5.43. Both of them send notifications to remote nodes. However, the cluster root is able to send an infinite number of nodes whereas the classic client can send a message to a unique node only. In addition, the current method of the multicasting is accomplished with a blocking manner. At least, the counterpart, the asynchronous ones are required. Anyway, developers can invoke the blocking methods with a thread to make it non-blocking. On the other hand, the current sample is not a practical application. In a real world system, the cluster root resides on a server or a peer. Thus, the multicasting is driven by a remote client or peer. Thus, the blocking methods are invoked by the threads of the types of NotificationQueue or RequestQueue in the pattern DWC. Thus, the blocking methods are usually called asynchronously. If so, it is not necessary to implement non-blocking methods for multicasting APIs. This is the way to transform a blocking method to a non-blocking one indirectly.

You can see the example in Section 2.6.4 of this chapter. In List 7.32 of ShutdownChildrenAdminNotificationThread.java, which is an instance of NotificationQueue, when a remote shutdown notification from the administrator, the notification queue invokes the blocking method, broadcastShutdownNotify(), of ClusterRoot (List 7.2) at Line 31~32. With the support of NotificationQueue, the blocking method runs asynchronously such that it is unnecessary to implement a non-blocking method for the broadcasting.

2.3 MN: Multicasting Notifier on CR

According to Table 7.6, in addition to SD, the pattern of CR utilizes two new patterns, i.e., MN (Multicasting Notifier) and MR (Multicasting Reader). Both of them are used to program the six multicasting APIs rapidly.

The MN is made up with three components, i.e., the data, the notification and the notification creator. On the other hand, the MR is composed of four components, including the data, the request, the response and the request creator. For both of the patterns, the component of the data is always enclosed by the messages. For MN, the data is enclosed in the notification whereas the data is placed into the request for MR.

2.3.1 MN for RootIPAddressBroadcastNotification: rootIPBroadcastNotifier

The first case utilizing MN, rootIPBroadcastNotifier, is the one that sends the root IP address to each child within the cluster. Each child needs to get the IP address such

that they can send messages to the root when necessary. Table 7.7 summarizes the pattern.

Pattern: MN (Multicasting Notifier)		
Class	ClusterRootBroadcastNotifier<IPAddress, RootIPAddressBroadcastNotification, RootIPAddressBroadcastNotificationCreator>	
Singleton	No	
Method	void notify(IPAddress data, int rootBranchCount, int subBranchCount)	
	void dispose()	
Enclosed Pattern	SO (Serialized Object)	IPAddress
	BN (Broadcast Notification)	RootIPAddressBroadcastNotification
	BNC (Broadcast Notification Creator)	RootIPAddressBroadcastNotificationCreator
Employed API	Serializable	
	MulticastMessage	
	ClusterRootBroadcastNotifier<Data extends Serializable, Notification extends MulticastMessage, NotificationCreator extends RootBroadcastNotificationCreatable<Notification, Data>>	
	RootBroadcastNotificationCreatable<Notification extends MulticastMessage, NotificationData extends Serializable>	

Table 7.7 The pattern of MN for RootIPAddressBroadcastNotification

The data of rootIPBroadcastNotifier is shown in List 7.3. According to the code, it is just a typical object to encapsulate data to be broadcast. Since it is necessary to be transmitted over the Internet, the data, IPAddress, ought to implement the interface, Serializable from JDK. To be general, the data is believed to be written in a simple pattern, Serialized Object (SO).

```
1   package com.greatfree.util;
2
3   import java.io.Serializable;
4
5   /*
6    * The IP address information is enclosed in the class. It is usually transmitted from the registry server to a
7    * cluster root. So, it must implement the interface of Serializable. 05/08/2017, Bing Li
8    */
9
10  // Created: 05/08/2017, Bing Li
11  public class IPAddress implements Serializable
12  {
13      private static final long serialVersionUID = -3975747063145993736L;
14
15      private String key;
16      private String ip;
17      private int port;
18
19      public IPAddress(String key, String ip, int port)
20      {
21          this.key = key;
22          this.ip = ip;
23          this.port = port;
24      }
25
26      public String getKey()
27      {
28          return this.key;
29      }
30
31      public String getIP()
32      {
33          return this.ip;
34      }
35
36      public int getPort()
37      {
38          return this.port;
39      }
40  }
```

List 7.3 The code of IPAddress.java

The notification of RootIPAddressBroadcastNotification is shown in List 7.4. Its parent class is MultiMessage rather than ServerMessage, which we use previously. The notification has two constructors. You do not need to care about the exact senses of those parameters in the construct if you program applications rapidly. But I would like to give you a brief introduction.

First of all, each constructor has a unique key. This is not necessary in our case. But in practice, it might become useful. For example, you can identify the state of a specific child after it receives one specific request according to the key. So I just leave the key in the messages for potential usage in particular applications.

As mentioned before, the purpose to design such a notification is to enclose the instance of IPAddress so as to send the data to all of the children within the cluster. So its constructor should have such a parameter as well.

One of the constructor has one additional parameter, HashMap<String, IPAddress> childrenServers). The constructor is employed when the scale of the cluster is larger than the value of the root branch count at least. For example, if the number of nodes in a cluster is only four whereas the value of the root branch count is five. It is not necessary to employ the constructor that has the parameter of HashMap<String, IPAddress>.

For the above considerations, it is reasonable to define the notification in a pattern, Broadcast Notification (BN).

```
1    package com.greatfree.cluster.message;
2
3    import java.util.HashMap;
4
5    import com.greatfree.message.MulticastMessage;
6    import com.greatfree.util.IPAddress;
7
8    /*
9     * The notification is a message to be sent to distributed nodes about the root IP address in the
10    * manner of broadcasting. 05/08/2017, Bing Li
11    */
12
13   // Created: 05/20/2017, Bing Li
14   public class RootIPAddressBroadcastNotification extends MulticastMessage
15   {
16       private static final long serialVersionUID = -3404599339423965092L;
17
18       private IPAddress rootAddress;
19
20       /*
21        * The constructor is used when the scale of the cluster is even smaller than the value of the
22        * root branch count. 06/14/2017, Bing Li
23        */
24       public RootIPAddressBroadcastNotification(String key, IPAddress rootAddress)
25       {
26           super(ClusterMessageType.ROOT_IPADDRESS_BROADCAST_NOTIFICATION, key);
27           this.rootAddress = rootAddress;
28       }
29
30       /*
31        * The constructor is used when the scale of the cluster is even larger than the value of the
32        * root branch count. 06/14/2017, Bing Li
33        */
34       public RootIPAddressBroadcastNotification(String key, HashMap<String, IPAddress> childrenServers,
35           IPAddress rootAddress)
36       {
37           super(ClusterMessageType.ROOT_IPADDRESS_BROADCAST_NOTIFICATION, key,
```

```
38                  childrenServers);
39              this.rootAddress = rootAddress;
40          }
41
42          public IPAddress getRootAddress()
43          {
44              return this.rootAddress;
45          }
46      }
```

List 7.4 The code of RootIPAddressBroadcastNotification.java

The notification creator, RootIPAddressBroadcastNotificationCreator, is shown in List 7.5, which is used to create new instances of messages when performing multicasting. RootBroadcastNotificationCreatable is the interface you need to implement for the creator. To be general, the structure is named as a pattern Broadcast Notification Creator (BNC).

```
1       package com.greatfree.cluster.message.root;
2
3       import java.util.HashMap;
4
5       import com.greatfree.cluster.message.RootIPAddressBroadcastNotification;
6       import com.greatfree.message.RootBroadcastNotificationCreatable;
7       import com.greatfree.util.IPAddress;
8       import com.greatfree.util.Tools;
9
10      /*
11       * The creator that generates the notification that is sent to distributed nodes broadcastly
12       * in a cluster. 05/08/2017, Bing Li
13       */
14
15      // Created: 05/20/2017, Bing Li
16      public class RootIPAddressBroadcastNotificationCreator implements RootBroadcastNotificationCreatable
17          <RootIPAddressBroadcastNotification, IPAddress>
18      {
19          /*
20           * The constructor is used to create notifications when the scale of the cluster is even larger than the value
21           * of the root branch count. 06/14/2017, Bing Li
22           */
23          @Override
24          public RootIPAddressBroadcastNotification createInstanceWithChildren(IPAddress message,
25              HashMap<String, IPAddress> childrenMap)
26          {
27              return new RootIPAddressBroadcastNotification(Tools.generateUniqueKey(), childrenMap, message);
28          }
29
30          /*
31           * The method is used to create notifications when the scale of the cluster is even smaller than the value of
32           * the root branch count. 06/14/2017, Bing Li
33           */
34          @Override
35          public RootIPAddressBroadcastNotification createInstanceWithoutChildren(IPAddress message)
36          {
37              return new RootIPAddressBroadcastNotification(Tools.generateUniqueKey(), message);
38          }
39      }
```

List 7.5 The code of RootIPAddressBroadcastNotificationCreator.java

2.3.2 MN for HelloWorldBroadcastNotification: helloWorldBroadcastNotifier

Since the program in the chapter is not practical, I just show you some contrived messages within the cluster. However, you can CPR them when implementing an application in the real world. Table 7.8 presents the pattern.

Pattern: MN (Multicasting Notifier)		
Class	ClusterRootBroadcastNotifier<HelloWorld, HelloWorldBroadcastNotification, HelloWorldBroadcastNotificationCreator>	
Singleton	No	
Method	void notify(HelloWorld data, int rootBranchCount, int subBranchCount)	
	void dispose()	
Enclosed Pattern	SO (Serialized Object)	HelloWorld
	BN (Broadcast Notification)	HelloWorldBroadcastNotification
	BNC (Broadcast Notification Creator)	HelloWorldBroadcastNotificationCreator
Employed API	Serializable	
	MulticastMessage	
	ClusterRootBroadcastNotifier<Data extends Serializable, Notification extends MulticastMessage, NotificationCreator extends RootBroadcastNotificationCreatable<Notification, Data>>	
	RootBroadcastNotificationCreatable<Notification extends MulticastMessage, NotificationData extends Serializable>	

Table 7.8 The pattern of MN for HelloWorldBroadcastNotification

The data to be broadcast is shown in List 7.6. The same as the data of IPAddress in List 7.3, HelloWorld should implement the interface Serializable as well such that it can be transmitted over the network. To be simplified, the data component of the MN, helloWorldBroadcastNotifier, contains one String only.

```
1    package com.greatfree.cluster;
2
3    import java.io.Serializable;
4
5    /*
6     * The testing data to be sent to distributed nodes in the multicasting manner. 05/08/2017, Bing Li
7     */
8
9    // Created: 05/08/2017, Bing Li
10   public class HelloWorld implements Serializable
11   {
12       private static final long serialVersionUID = -3366357589324485114L;
13
14       private String helloWorld;
15
16       public HelloWorld(String helloWorld)
17       {
18           this.helloWorld = helloWorld;
19       }
20
21       public String getHelloWorld()
22       {
23           return this.helloWorld;
24       }
25   }
```

List 7.6 The code of HelloWorld.java

The notification of helloWorldBroadcastNotifier is called, HelloWorldBroadcastNotification, which is shown in List 7.7. You need to note that its structure is identical to the one of RootIPAddressBroadcastNotification.java in List 7.4.

```
1    package com.greatfree.cluster.message;
2
3    import java.util.HashMap;
4
5    import com.greatfree.cluster.HelloWorld;
6    import com.greatfree.message.MulticastMessage;
7    import com.greatfree.util.IPAddress;
8
9    /*
10    * The notification is a testing message to be sent to distributed nodes in the manner of
11    * broadcasting. 05/08/2017, Bing Li
12    */
```

```
13
14  // Created: 05/08/2017, Bing Li
15  public class HelloWorldBroadcastNotification extends MulticastMessage
16  {
17      private static final long serialVersionUID = 6393056123607868530L;
18
19      private HelloWorld hw;
20
21      /*
22       * The constructor is used when the scale of the cluster is even smaller than the value of the
23       * root branch count. 06/14/2017, Bing Li
24       */
25      public HelloWorldBroadcastNotification(String key, HelloWorld hw)
26      {
27          super(ClusterMessageType.HELLO_WORLD_BROADCAST_NOTIFICATION, key);
28          this.hw = hw;
29      }
30
31      /*
32       * The constructor is used when the scale of the cluster is even larger than the value of
33       * the root branch count. 06/14/2017, Bing Li
34       */
35      public HelloWorldBroadcastNotification(String key, HashMap<String, IPAddress> childrenServers,
36          HelloWorld hw)
37      {
38          super(ClusterMessageType.HELLO_WORLD_BROADCAST_NOTIFICATION, key,
39              childrenServers);
40          this.hw = hw;
41      }
42
43      public HelloWorld getHelloWorld()
44      {
45          return this.hw;
46      }
47  }
```

List 7.7 The code of HelloWorldBroadcastNotification.java

The creator, HelloWorldBroadcastNotificationCreator, is shown in List 7.8.

```
1   package com.greatfree.cluster.message.root;
2
3   import java.util.HashMap;
4
5   import com.greatfree.cluster.HelloWorld;
6   import com.greatfree.cluster.message.HelloWorldBroadcastNotification;
7   import com.greatfree.message.RootBroadcastNotificationCreatable;
8   import com.greatfree.util.IPAddress;
9   import com.greatfree.util.Tools;
10
11  /*
12   * The creator that generates the notification that is sent to distributed nodes broadcastly
13   * in a cluster. 05/08/2017, Bing Li
14   */
15
16  // Created: 05/08/2017, Bing Li
17  public class HelloWorldBroadcastNotificationCreator implements RootBroadcastNotificationCreatable
18      <HelloWorldBroadcastNotification, HelloWorld>
19  {
20      /*
21       * The constructor is used to create notifications when the scale of the cluster is even larger than the value
22       * of the root branch count. 06/14/2017, Bing Li
23       */
24      @Override
25      public HelloWorldBroadcastNotification createInstanceWithChildren(HelloWorld message,
26          HashMap<String, IPAddress> childrenMap)
27      {
28          return new HelloWorldBroadcastNotification(Tools.generateUniqueKey(), childrenMap, message);
29      }
30
31      /*
32       * The method is used to create notifications when the scale of the cluster is even smaller than the value of
33       * the root branch count. 06/14/2017, Bing Li
34       */
```

```
35        @Override
36        public HelloWorldBroadcastNotification createInstanceWithoutChildren(HelloWorld message)
37        {
38            return new HelloWorldBroadcastNotification(Tools.generateUniqueKey(), message);
39        }
40    }
```

List 7.8 The code of HelloWorldBroadcastNotificationCreator.java

2.3.3 MN for HelloWorldAnycastNotification: helloWorldAnycastNotifier

The data, HelloWorld, of the MN, helloWorldAnycastNotifier, is the same as the one for HelloWorldBroadcastNotification, as shown in List 7.6. Table 7.9 illustrates the pattern.

Pattern: MN (Multicasting Notifier)		
Class	ClusterRootAnycastNotifier<HelloWorld, HelloWorldAnycastNotification, HelloWorldAnycastNotificationCreator>	
Singleton	No	
Method	void notify(HelloWorld data, int rootBranchCount, int subBranchCount)	
	void dispose()	
Enclosed Pattern	SO (Serialized Object)	HelloWorld
	BN (Broadcast Notification)	HelloWorldAnycastNotification
	BNC (Broadcast Notification Creator)	HelloWorldAnycastNotificationCreator
Employed API	Serializable	
	MulticastMessage	
	ClusterRootAnycastNotifier<Data extends Serializable, Notification extends MulticastMessage, NotificationCreator extends RootBroadcastNotificationCreatable<Notification, Data>>	
	RootBroadcastNotificationCreatable<Notification extends MulticastMessage, NotificationData extends Serializable>	

Table 7.9 The pattern of MN for HelloWorldAnycastNotification

The notification, HelloWorldAnycastNotification, is shown in List 7.9.

```
1     package com.greatfree.cluster.message;
2
3     import java.util.HashMap;
4
5     import com.greatfree.cluster.HelloWorld;
6     import com.greatfree.message.MulticastMessage;
7     import com.greatfree.util.IPAddress;
8
9     /*
10     * The notification is a testing message to be sent to distributed data in the manner
11     * of anycasting. 05/08/2017, Bing Li
12     */
13
14    // Created: 05/19/2017, Bing Li
15    public class HelloWorldAnycastNotification extends MulticastMessage
16    {
17        private static final long serialVersionUID = -8596454328337543079L;
18
19        private HelloWorld hw;
20
21        /*
22         * The constructor is used when the scale of the cluster is even smaller than the value of the
23         * root branch count. 06/14/2017, Bing Li
24         */
25        public HelloWorldAnycastNotification(String key, HelloWorld hw)
26        {
27            super(ClusterMessageType.HELLO_WORLD_ANYCAST_NOTIFICATION, key);
28            this.hw = hw;
29        }
30
31        /*
32         * The constructor is used when the scale of the cluster is even larger than the value of the
33         * root branch count. 06/14/2017, Bing Li
```

```
34          */
35          public HelloWorldAnycastNotification(String key, HashMap<String, IPAddress> childrenServers,
36              HelloWorld hw)
37          {
38              super(ClusterMessageType.HELLO_WORLD_ANYCAST_NOTIFICATION, key,
39                  childrenServers);
40              this.hw = hw;
41          }
42
43          public HelloWorld getHelloWorld()
44          {
45              return this.hw;
46          }
47      }
```

List 7.9 The code of HelloWorldAnycastNotification.java

The creator, HelloWorldAnycastNotificationCreator, is shown in List 7.10.

```
1       package com.greatfree.cluster.message.root;
2
3       import java.util.HashMap;
4
5       import com.greatfree.cluster.HelloWorld;
6       import com.greatfree.cluster.message.HelloWorldAnycastNotification;
7       import com.greatfree.message.RootBroadcastNotificationCreatable;
8       import com.greatfree.util.IPAddress;
9       import com.greatfree.util.Tools;
10
11      /*
12       * The creator that generates the notification that is sent to distributed nodes anycastly
13       * in a cluster. 05/08/2017, Bing Li
14       */
15
16      // Created: 05/19/2017, Bing Li
17      public class HelloWorldAnycastNotificationCreator implements RootBroadcastNotificationCreatable
18          <HelloWorldAnycastNotification, HelloWorld>
19      {
20          /*
21           * The constructor is used to create notifications when the scale of the cluster is even larger than the value
22           * of the root branch count. 06/14/2017, Bing Li
23           */
24          @Override
25          public HelloWorldAnycastNotification createInstanceWithChildren(HelloWorld message, HashMap<String,
26              IPAddress> childrenMap)
27          {
28              return new HelloWorldAnycastNotification(Tools.generateUniqueKey(), childrenMap, message);
29          }
30
31          /*
32           * The method is used to create notifications when the scale of the cluster is even smaller than the value of
33           * the root branch count. 06/14/2017, Bing Li
34           */
35          @Override
36          public HelloWorldAnycastNotification createInstanceWithoutChildren(HelloWorld message)
37          {
38              return new HelloWorldAnycastNotification(Tools.generateUniqueKey(), message);
39          }
40      }
```

List 7.10 The code of HelloWorldAnycastNotificationCreator.java

2.3.4 MN for HelloWorldUnicastNotification: helloWorldUnicastNotifier

The data of the MN, helloWorldUnicastNotifier (Table 7.10), is also HelloWorld, as shown in List 7.6.

Another component, the notification, HelloWorldUnicastNotification, is shown in List 7.11.

Pattern: MN (Multicasting Notifier)		
Class	ClusterRootUnicastNotifier<HelloWorld, HelloWorldUnicastNotification, HelloWorldUnicastNotificationCreator>	
Singleton	No	
Method	void notify(HelloWorld data, int rootBranchCount, int subBranchCount)	
	void dispose()	
Enclosed Pattern	SO (Serialized Object)	HelloWorld
	BN (Broadcast Notification)	HelloWorldUnicastNotification
	BNC (Broadcast Notification Creator)	HelloWorldUnicastNotificationCreator
Employed API	Serializable	
	MulticastMessage	
	ClusterRootUnicastNotifier<Data extends Serializable, Notification extends MulticastMessage, NotificationCreator extends RootBroadcastNotificationCreatable<Notification, Data>>	
	RootBroadcastNotificationCreatable<Notification extends MulticastMessage, NotificationData extends Serializable>	

Table 7.10 The pattern of MN for HelloWorldUnicastNotification

```
1   package com.greatfree.cluster.message;
2
3   import java.util.HashMap;
4
5   import com.greatfree.cluster.HelloWorld;
6   import com.greatfree.message.MulticastMessage;
7   import com.greatfree.util.IPAddress;
8
9   /*
10   * The notification is a testing message to be sent to distributed data in the manner of
11   * unicasting. 05/19/2017, Bing Li
12   */
13
14  // Created: 05/19/2017, Bing Li
15  public class HelloWorldUnicastNotification extends MulticastMessage
16  {
17      private static final long serialVersionUID = 8129038077864608395L;
18
19      private HelloWorld hw;
20
21      /*
22       * The constructor is used when the scale of the cluster is even smaller than the value of the root
23       * branch count. 06/14/2017, Bing Li
24       */
25      public HelloWorldUnicastNotification(String key, HelloWorld hw)
26      {
27          super(ClusterMessageType.HELLO_WORLD_UNICAST_NOTIFICATION, key);
28          this.hw = hw;
29      }
30
31      /*
32       * The constructor is used when the scale of the cluster is even larger than the value of the root
33       * branch count. 06/14/2017, Bing Li
34       */
35      public HelloWorldUnicastNotification(String key, HashMap<String, IPAddress> childrenServers,
36          HelloWorld hw)
37      {
38          super(ClusterMessageType.HELLO_WORLD_UNICAST_NOTIFICATION, key,
39              childrenServers);
40          this.hw = hw;
41      }
42
43      public HelloWorld getHelloWorld()
44      {
45          return this.hw;
46      }
47  }
```

List 7.11 The code of HelloWorldUnicastNotification.java

The creator, HelloWorldUnicastNotificationCreator, for the notification, HelloWorldUnicastNotification, is shown in List 7.12.

```
1    package com.greatfree.cluster.message.root;
2
3    import java.util.HashMap;
4
5    import com.greatfree.cluster.HelloWorld;
6    import com.greatfree.cluster.message.HelloWorldUnicastNotification;
7    import com.greatfree.message.RootBroadcastNotificationCreatable;
8    import com.greatfree.util.IPAddress;
9    import com.greatfree.util.Tools;
10
11   /*
12    * The creator that generates the notification that is sent to distributed nodes unicastly
13    * in a cluster. 05/19/2017, Bing Li
14    */
15
16   // Created: 05/19/2017, Bing Li
17   public class HelloWorldUnicastNotificationCreator implements RootBroadcastNotificationCreatable
18       <HelloWorldUnicastNotification, HelloWorld>
19   {
20       /*
21        * The constructor is used to create notifications when the scale of the cluster is even larger than
22        * the value of the root branch count. 06/14/2017, Bing Li
23        */
24       @Override
25       public HelloWorldUnicastNotification createInstanceWithChildren(HelloWorld message, HashMap<String,
26           IPAddress> childrenMap)
27       {
28           return new HelloWorldUnicastNotification(Tools.generateUniqueKey(), childrenMap, message);
29       }
30
31       /*
32        * The method is used to create notifications when the scale of the cluster is even smaller than the value of
33        * the root branch count. 06/14/2017, Bing Li
34        */
35       @Override
36       public HelloWorldUnicastNotification createInstanceWithoutChildren(HelloWorld message)
37       {
38           return new HelloWorldUnicastNotification(Tools.generateUniqueKey(), message);
39       }
40   }
```

List 7.12 The code of HelloWorldUnicastNotificationCreator.java

2.3.5 MN for ShutdownChildrenBroadcastNotification: shutdownChildrenBroadcastNotifier

Again, as a contrived sample system, the notification for broadcasting is not real. You are expected to revise the code when facing with the real world cases.

The data component for the MN, shutdownChildrenBroadcastNotifier, is still HelloWorld, as shown in List 7.6. However, it only plays the role of a placeholder to fulfill the requirement to define the MN. Table 7.11 summarizes the pattern.

The notification of the MN is ShutdownChildrenBroadcastNotification, which is shown in List 7.13.

```
1    package com.greatfree.cluster.message;
2
3    import java.util.HashMap;
4
5    import com.greatfree.message.MulticastMessage;
6    import com.greatfree.util.IPAddress;
```

```
7
8       /*
9        * After receiving the administration message of shutting down from the administrator, the root sends a
10       * broadcast notification to its children. 05/19/2017, Bing Li
11       */
12
13      // Created: 05/19/2017, Bing Li
14      public class ShutdownChildrenBroadcastNotification extends MulticastMessage
15      {
16          private static final long serialVersionUID = -8645482803978813301L;
17
18          /*
19           * The constructor is used when the scale of the cluster is even smaller than the value of the root branch
20           * count. 06/14/2017, Bing Li
21           */
22          public ShutdownChildrenBroadcastNotification(String key)
23          {
24              super(ClusterMessageType.SHUTDOWN_CHILDREN_BROADCAST_NOTIFICATION, key);
25          }
26
27          /*
28           * The constructor is used when the scale of the cluster is even larger than the value of the root branch
29           * count. 06/14/2017, Bing Li
30           */
31          public ShutdownChildrenBroadcastNotification(String key, HashMap<String, IPAddress> childrenServers)
32          {
33              super(ClusterMessageType.SHUTDOWN_CHILDREN_BROADCAST_NOTIFICATION, key,
34                  childrenServers);
35          }
36      }
```

List 7.13 The code of ShutdownChildrenBroadcastNotification.java

Pattern: MN (Multicasting Notifier)		
Class	ClusterRootBroadcastNotifier<HelloWorld, ShutdownChildrenBroadcastNotification, ShutdownChildrenBroadcastNotificationCreator>	
Singleton	No	
Method	void notify(HelloWorld data, int rootBranchCount, int subBranchCount)	
	void dispose()	
Enclosed Pattern	SO (Serialized Object)	HelloWorld
	BN (Broadcast Notification)	ShutdownChildrenBroadcastNotification
	BNC (Broadcast Notification Creator)	ShutdownChildrenBroadcastNotificationCreator
Employed API	Serializable	
	MulticastMessage	
	ClusterRootBroadcastNotifier<Data extends Serializable, Notification extends MulticastMessage, NotificationCreator extends RootBroadcastNotificationCreatable<Notification, Data>>	
	RootBroadcastNotificationCreatable<Notification extends MulticastMessage, NotificationData extends Serializable>	

Table 7.11 The pattern MN for ShutdownChildrenBroadcastNotification

The notification creator for the MN is named as ShutdownChildrenBroadcastNotificationCreator, which is shown in List 7.14.

```
1       package com.greatfree.cluster.message.root;
2
3       import java.util.HashMap;
4
5       import com.greatfree.cluster.HelloWorld;
6       import com.greatfree.cluster.message.ShutdownChildrenBroadcastNotification;
7       import com.greatfree.message.RootBroadcastNotificationCreatable;
8       import com.greatfree.util.IPAddress;
9       import com.greatfree.util.Tools;
10
11      // Created: 05/19/2017, Bing Li
12      public class ShutdownChildrenBroadcastNotificationCreator implements
13          RootBroadcastNotificationCreatable<ShutdownChildrenBroadcastNotification, HelloWorld>
14      {
15          /*
16           * The constructor is used to create notifications when the scale of the cluster is even larger than the value
17           * of the root branch count. 06/14/2017, Bing Li
18           */
```

```
19        @Override
20        public ShutdownChildrenBroadcastNotification createInstanceWithChildren(HelloWorld message,
21            HashMap<String, IPAddress> childrenMap)
22        {
23            return new ShutdownChildrenBroadcastNotification(Tools.generateUniqueKey(), childrenMap);
24        }
25
26        /*
27         * The method is used to create notifications when the scale of the cluster is even smaller than the value of
28         * the root branch count. 06/14/2017, Bing Li
29         */
30        @Override
31        public ShutdownChildrenBroadcastNotification createInstanceWithoutChildren(HelloWorld message)
32        {
33            return new ShutdownChildrenBroadcastNotification(Tools.generateUniqueKey());
34        }
35    }
```

List 7.14 The code of ShutdownChildrenBroadcastNotificationCreator.java

2.4 MR: Multicasting Reader on CR

As mentioned before, the pattern of Multicasting Reader (MR) is made up with four components, i.e., the data, the request, the response and the request creator. The CR employs three MRs, including helloWorldBroadcastReader, helloWorldAnycastReader and helloWorldUnicastReader.

2.4.1 MR for HelloWorldBroadcastRequest: helloWorldBroadcastReader

The data for the MR, helloWorldBroadcastReader, is still HelloWorld, as shown in List 7.6.

The request, HelloWorldBroadcastRequest, is shown in List 7.15. The request has two constructors as well. Compared with the notification of MN, the constructor of the message has one additional parameter, collaboratorKey (String). As a highly concurrent system, it is possible for the root to perform multiple request multicasting concurrently. Since responses are sent back from children in the cluster, it is necessary to differentiate the counterpart, the request, of one particular response. So it is mandatory to create one unique key for each request multicasting. The key is called the collaborator key. To be clear, the request is believed to be simple pattern, Broadcast Request (BREQ).

```
1     package com.greatfree.cluster.message;
2
3     import java.util.HashMap;
4
5     import com.greatfree.cluster.HelloWorld;
6     import com.greatfree.message.ClusterBroadcastRequest;
7     import com.greatfree.util.IPAddress;
8
9     /*
10     * The request is broadcast to each of the distributed node in a cluster and then each of the node
11     * needs to respond to the root individually. The responses are merged together as the final
12     * response. 05/20/2017, Bing Li
13     */
14
15    // Created: 05/20/2017, Bing Li
16    public class HelloWorldBroadcastRequest extends ClusterBroadcastRequest
17    {
18        private static final long serialVersionUID = 7467117351318715616L;
19
```

```
20          private HelloWorld hw;
21
22          /*
23           * The constructor is used when the scale of the cluster is even smaller than the value of the root
24           * branch count. 06/14/2017, Bing Li
25           */
26          public HelloWorldBroadcastRequest(HelloWorld hw, String key, String collaboratorKey)
27          {
28              super(ClusterMessageType.HELLO_WORLD_BROADCAST_REQUEST, key, collaboratorKey);
29              this.hw = hw;
30          }
31
32          /*
33           * The constructor is used when the scale of the cluster is even larger than the value of the root
34           * branch count. 06/14/2017, Bing Li
35           */
36          public HelloWorldBroadcastRequest(HelloWorld hw, String key, String collaboratorKey, HashMap<String,
37              IPAddress> childrenServerMap)
38          {
39              super(ClusterMessageType.HELLO_WORLD_BROADCAST_REQUEST, key, collaboratorKey,
40                  childrenServerMap);
41              this.hw = hw;
42          }
43
44          public HelloWorld getHelloWorld()
45          {
46              return this.hw;
47          }
48      }
```

List 7.15 The code of HelloWorldBroadcastRequest.java

The response, HelloWorldBroadcastResponse, for the MR, helloWorldBroadcastReader, is shown in List 7.16. You should pay attention to the parameter, collaboratorKey (String), in the constructor. So, the response is believed to be written in a pattern of Broadcast Response (BRES).

```
1       package com.greatfree.cluster.message;
2
3       import com.greatfree.cluster.HelloWorld;
4       import com.greatfree.message.ClusterBroadcastResponse;
5
6       /*
7        * The response is the counterpart of HelloWorldBroadcastRequest. 05/20/2017, Bing Li
8        */
9
10      // Created: 05/20/2017, Bing Li
11      public class HelloWorldBroadcastResponse extends ClusterBroadcastResponse
12      {
13          private static final long serialVersionUID = 3885107291746728630L;
14
15          private HelloWorld hw;
16
17          public HelloWorldBroadcastResponse(HelloWorld hw, String key, String collaboratorKey)
18          {
19              super(ClusterMessageType.HELLO_WORLD_BROADCAST_RESPONSE, key,
20                  collaboratorKey);
21              this.hw = hw;
22          }
23
24          public HelloWorld getHelloWorld()
25          {
26              return this.hw;
27          }
28      }
```

List 7.16 The code of HelloWorldBroadcastResponse.java

The creator, HelloWorldBroadcastRequestCreator, is the fourth component for the MR. Its code is shown in List 7.17. To be consistent, the creator is placed in the

pattern of BRC (Broadcast Request Creator). Table 7.12 summarizes the pattern of the MR for HelloWorldBroadcastRequest.

Pattern: MR (Multicasting Reader)		
Class	ClusterRootBroadcastReader<HelloWorld, HelloWorldBroadcastRequest, HelloWorldBroadcastResponse, HelloWorldBroadcastRequestCreator>	
Singleton	No	
Method	Map<String, HelloWorldBroadcastResponse> read(HelloWorld data, int rootBranchCount, int subBranchCount)	
	void dispose()	
	void notifyResponseReceived(HelloWorldBroadcastResponse response)	
Enclosed Pattern	SO (Serialized Object)	HelloWorld
	BREQ (Broadcast Request)	HelloWorldBroadcastRequest
	BRES (Broadcast Response)	HelloWorldBroadcastResponse
	BRC (Broadcast Request Creator)	HelloWorldBroadcastRequestCreator
Employed API	Serializable	
	ClusterBroadcastRequest	
	ClusterBroadcastResponse	
	ClusterRootBroadcastReader<Data extends Serializable, Request extends ClusterBroadcastRequest, Response extends ClusterBroadcastResponse, RequestCreator extends RootBroadcastRequestCreatable<Request, Data>>	
	RootBroadcastRequestCreatable<Request extends ClusterBroadcastRequest, ObjectedData extends Serializable>	

Table 7.12 The pattern of MR for HelloWorldBroadcastRequest

```
1   package com.greatfree.cluster.message.root;
2
3   import java.util.HashMap;
4
5   import com.greatfree.cluster.HelloWorld;
6   import com.greatfree.cluster.message.HelloWorldBroadcastRequest;
7   import com.greatfree.message.RootBroadcastRequestCreatable;
8   import com.greatfree.util.IPAddress;
9   import com.greatfree.util.Tools;
10
11  /*
12   * The creator that generates the request that is sent to distributed nodes broadcastly in
13   * a cluster. 05/20/2017, Bing Li
14   */
15
16  // Created: 05/20/2017, Bing Li
17  public class HelloWorldBroadcastRequestCreator implements RootBroadcastRequestCreatable
18      <HelloWorldBroadcastRequest, HelloWorld>
19  {
20      /*
21       * The constructor is used to create requests when the scale of the cluster is even larger than the value of
22       * the root branch count. 06/14/2017, Bing Li
23       */
24      @Override
25      public HelloWorldBroadcastRequest createInstanceWithChildren(HelloWorld t, String collaboratorKey,
26          HashMap<String, IPAddress> childrenMap)
27      {
28          return new HelloWorldBroadcastRequest(t, Tools.generateUniqueKey(), collaboratorKey,
29              childrenMap);
30      }
31
32      /*
33       * The method is used to create request when the scale of the cluster is even smaller than the value of the
34       * root branch count. 06/14/2017, Bing Li
35       */
36      @Override
37      public HelloWorldBroadcastRequest createInstanceWithoutChildren(HelloWorld t, String collaboratorKey)
38      {
39          return new HelloWorldBroadcastRequest(t, Tools.generateUniqueKey(), collaboratorKey);
40      }
41  }
```

List 7.17 The code of HelloWorldBroadcastRequestCreator.java

2.4.2 MR for HelloWorldAnycastRequest: helloWorldAnycastReader

The pattern is presented in Table 7.13.

Pattern: MR (Multicasting Reader)		
Class	ClusterRootAnycastReader<HelloWorld, HelloWorldAnycastRequest, HelloWorldAnycastResponse, HelloWorldAnycastRequestCreator>	
Singleton	No	
Method	Map<String, HelloWorldAnycastResponse> read(HelloWorld data, int rootBranchCount, int subBranchCount)	
	void dispose()	
	void notifyResponseReceived(HelloWorldAnycastResponse response)	
Enclosed Pattern	SO (Serialized Object)	HelloWorld
	BREQ (Broadcast Request)	HelloWorldAnycastRequest
	BRES (Broadcast Response)	HelloWorldAnycastResponse
	BRC (Broadcast Request Creator)	HelloWorldAnycastRequestCreator
Employed API	Serializable	
	ClusterBroadcastRequest	
	ClusterBroadcastResponse	
	ClusterRootAnycastReader<Data extends Serializable, Request extends ClusterBroadcastRequest, Response extends ClusterBroadcastResponse, RequestCreator extends RootBroadcastRequestCreatable<Request, Data>>	
	RootBroadcastRequestCreatable<Request extends ClusterBroadcastRequest, ObjectedData extends Serializable>	

Table 7.13 The pattern of MR for HelloWorldAnycastRequest

The data for the MR, helloWorldAnycastReader, is HelloWorld, as shown in List 7.6.

The request, HelloWorldAnycastRequest, is shown in List 7.18.

```
1    package com.greatfree.cluster.message;
2
3    import java.util.HashMap;
4
5    import com.greatfree.cluster.HelloWorld;
6    import com.greatfree.message.ClusterBroadcastRequest;
7    import com.greatfree.util.IPAddress;
8
9    /*
10    * The request is anycast to any of the distributed nodes in a cluster and then the node that receives the
11    * request needs to respond to the root individually. The responses are merged together as the final
12    * response. 05/20/2017, Bing Li
13    */
14
15   // Created: 05/21/2017, Bing Li
16   public class HelloWorldAnycastRequest extends ClusterBroadcastRequest
17   {
18       private static final long serialVersionUID = -3645189237784363115L;
19
20       private HelloWorld hw;
21
22       /*
23        * The constructor is used when the scale of the cluster is even smaller than the value of the root
24        * branch count. 06/14/2017, Bing Li
25        */
26       public HelloWorldAnycastRequest(HelloWorld hw, String key, String collaboratorKey)
27       {
28           super(ClusterMessageType.HELLO_WORLD_ANYCAST_REQUEST, key, collaboratorKey);
29           this.hw = hw;
30       }
31
32       /*
33        * The constructor is used when the scale of the cluster is even larger than the value of the root
34        * branch count. 06/14/2017, Bing Li
35        */
36       public HelloWorldAnycastRequest(HelloWorld hw, String key, String collaboratorKey, HashMap<String,
37           IPAddress> childrenServerMap)
38       {
39           super(ClusterMessageType.HELLO_WORLD_ANYCAST_REQUEST, key, collaboratorKey,
40               childrenServerMap);
```

```
41            this.hw = hw;
42        }
43
44        public HelloWorld getHelloWorld()
45        {
46            return this.hw;
47        }
48    }
```

List 7.18 The code of HelloWorldAnycastRequest.java

The response, HelloWorldAnycastResponse, is shown in List 7.19.

```
1     package com.greatfree.cluster.message;
2
3     import com.greatfree.cluster.HelloWorld;
4     import com.greatfree.message.ClusterBroadcastResponse;
5
6     /*
7      * The response is the counterpart of HelloWorldAnycastRequest. 05/20/2017, Bing Li
8      */
9
10    // Created: 05/21/2017, Bing Li
11    public class HelloWorldAnycastResponse extends ClusterBroadcastResponse
12    {
13        private static final long serialVersionUID = -4984663742943784120L;
14
15        private HelloWorld hw;
16
17        public HelloWorldAnycastResponse(HelloWorld hw, String key, String collaboratorKey)
18        {
19            super(ClusterMessageType.HELLO_WORLD_ANYCAST_RESPONSE, key, collaboratorKey);
20            this.hw = hw;
21        }
22
23        public HelloWorld getHelloWorld()
24        {
25            return this.hw;
26        }
27    }
```

List 7.19 The code of HelloWorldAnycastResponse.java

The creator, HelloWorldAnycastRequestCreator, is shown in List 7.20.

```
1     package com.greatfree.cluster.message.root;
2
3     import java.util.HashMap;
4
5     import com.greatfree.cluster.HelloWorld;
6     import com.greatfree.cluster.message.HelloWorldAnycastRequest;
7     import com.greatfree.message.RootBroadcastRequestCreatable;
8     import com.greatfree.util.IPAddress;
9     import com.greatfree.util.Tools;
10
11    /*
12     * The creator that generates the request that is sent to distributed nodes anycastly in
13     * a cluster. 05/20/2017, Bing Li
14     */
15
16    // Created: 05/21/2017, Bing Li
17    public class HelloWorldAnycastRequestCreator implements RootBroadcastRequestCreatable
18        <HelloWorldAnycastRequest, HelloWorld>
19    {
20        /*
21         * The constructor is used to create requests when the scale of the cluster is even larger than the value of
22         * the root branch count. 06/14/2017, Bing Li
23         */
24        @Override
25        public HelloWorldAnycastRequest createInstanceWithChildren(HelloWorld t, String collaboratorKey,
26            HashMap<String, IPAddress> childrenMap)
```

```
27          {
28              return new HelloWorldAnycastRequest(t, Tools.generateUniqueKey(), collaboratorKey, childrenMap);
29          }
30
31          /*
32           * The method is used to create request when the scale of the cluster is even smaller than the value of the
33           * root branch count. 06/14/2017, Bing Li
34           */
35          @Override
36          public HelloWorldAnycastRequest createInstanceWithoutChildren(HelloWorld t, String collaboratorKey)
37          {
38              return new HelloWorldAnycastRequest(t, Tools.generateUniqueKey(), collaboratorKey);
39          }
40      }
```

List 7.20 The code of HelloWorldAnycastRequestCreator.java

2.4.3 MR for HelloWorldUnicastRequest: helloWorldUnicatReader

The pattern is summed up in Table 7.14.

Pattern: MR (Multicasting Reader)		
Class	ClusterRootUnicastReader<HelloWorld, HelloWorldUnicastRequest, HelloWorldUnicastResponse, HelloWorldUnicastRequestCreator>	
Singleton	No	
Method	Map<String, HelloWorldUnicastResponse> read(HelloWorld data, int rootBranchCount, int subBranchCount)	
	void dispose()	
	void notifyResponseReceived(HelloWorldUnicastResponse response)	
Enclosed Pattern	SO (Serialized Object)	HelloWorld
	BREQ (Broadcast Request)	HelloWorldUnicastRequest
	BRES (Broadcast Response)	HelloWorldUnicastResponse
	BRC (Broadcast Request Creator)	HelloWorldUnicastRequestCreator
Employed API	Serializable	
	ClusterBroadcastRequest	
	ClusterBroadcastResponse	
	ClusterRootUnicastReader<Data extends Serializable, Request extends ClusterBroadcastRequest, Response extends ClusterBroadcastResponse, RequestCreator extends RootBroadcastRequestCreatable<Request, Data>>	
	RootBroadcastRequestCreatable<Request extends ClusterBroadcastRequest, ObjectedData extends Serializable>	

Table 7.14 The pattern of MR for HelloWorldUnicastRequest

The data of the MR, helloWorldUnicastReader, is shown in List 7.6.

The request, HelloWorldUnicastRequest, is shown in List 7.21.

```
1       package com.greatfree.cluster.message;
2
3       import java.util.HashMap;
4
5       import com.greatfree.cluster.HelloWorld;
6       import com.greatfree.message.ClusterBroadcastRequest;
7       import com.greatfree.util.IPAddress;
8
9       /*
10       * The request is unicast to any of the distributed nodes in a cluster and then the node that receives
11       * the request needs to respond to the root individually. The responses are merged together as the
12       * final response. 05/20/2017, Bing Li
13       */
14
15      // Created: 05/21/2017, Bing Li
16      public class HelloWorldUnicastRequest extends ClusterBroadcastRequest
17      {
18          private static final long serialVersionUID = -2317366858101531395L;
19
20          private HelloWorld hw;
```

```
21
22          /*
23           * The constructor is used when the scale of the cluster is even smaller than the value of the
24           * root branch count. 06/14/2017, Bing Li
25           */
26          public HelloWorldUnicastRequest(HelloWorld hw, String key, String collaboratorKey)
27          {
28              super(ClusterMessageType.HELLO_WORLD_UNICAST_REQUEST, key, collaboratorKey);
29              this.hw = hw;
30          }
31
32          /*
33           * The constructor is used when the scale of the cluster is even larger than the value of the
34           * root branch count. 06/14/2017, Bing Li
35           */
36          public HelloWorldUnicastRequest(HelloWorld hw, String key, String collaboratorKey,
37              HashMap<String, IPAddress> childrenServerMap)
38          {
39              super(ClusterMessageType.HELLO_WORLD_UNICAST_REQUEST, key, collaboratorKey,
40                  childrenServerMap);
41              this.hw = hw;
42          }
43
44          public HelloWorld getHelloWorld()
45          {
46              return this.hw;
47          }
48      }
```

List 7.21 The code of HelloWorldUnicastRequest.java

The response of the MR, HelloWorldUnicastResponse, is shown in List 7.22.

```
1       package com.greatfree.cluster.message;
2
3       import com.greatfree.cluster.HelloWorld;
4       import com.greatfree.message.ClusterBroadcastResponse;
5
6       /*
7        * The response is the counterpart of HelloWorldUnicastRequest. 05/20/2017, Bing Li
8        */
9
10      // Created: 05/21/2017, Bing Li
11      public class HelloWorldUnicastResponse extends ClusterBroadcastResponse
12      {
13          private static final long serialVersionUID = 7583663294458313886L;
14
15          private HelloWorld hw;
16
17          public HelloWorldUnicastResponse(HelloWorld hw, String key, String collaboratorKey)
18          {
19              super(ClusterMessageType.HELLO_WORLD_UNICAST_RESPONSE, key, collaboratorKey);
20              this.hw = hw;
21          }
22
23          public HelloWorld getHelloWorld()
24          {
25              return this.hw;
26          }
27      }
```

List 7.22 The code of HelloWorldUnicastResponse.java

The request creator, HelloWorldUnicastRequestCreator, is shown in List 7.23.

```
1       package com.greatfree.cluster.message.root;
2
3       import java.util.HashMap;
4
5       import com.greatfree.cluster.HelloWorld;
6       import com.greatfree.cluster.message.HelloWorldUnicastRequest;
```

```
7    import com.greatfree.message.RootBroadcastRequestCreatable;
8    import com.greatfree.util.IPAddress;
9    import com.greatfree.util.Tools;
10
11   /*
12    * The creator that generates the request that is sent to distributed nodes unicastly in
13    * a cluster. 05/20/2017, Bing Li
14    */
15
16   // Created: 05/21/2017, Bing Li
17   public class HelloWorldUnicastRequestCreator implements RootBroadcastRequestCreatable
18       <HelloWorldUnicastRequest, HelloWorld>
19   {
20       /*
21        * The constructor is used to create requests when the scale of the cluster is even larger than
22        * the value of the root branch count. 06/14/2017, Bing Li
23        */
24       @Override
25       public HelloWorldUnicastRequest createInstanceWithChildren(HelloWorld t, String collaboratorKey,
26           HashMap<String, IPAddress> childrenMap)
27       {
28           return new HelloWorldUnicastRequest(t, Tools.generateUniqueKey(), collaboratorKey, childrenMap);
29       }
30
31       /*
32        * The method is used to create request when the scale of the cluster is even smaller than
33        * the value of the root branch count. 06/14/2017, Bing Li
34        */
35       @Override
36       public HelloWorldUnicastRequest createInstanceWithoutChildren(HelloWorld t, String collaboratorKey)
37       {
38           return new HelloWorldUnicastRequest(t, Tools.generateUniqueKey(), collaboratorKey);
39       }
40   }
```

List 7.23 The code of HelloWorldUnicastRequestCreator.java

2.5 SD: Programming the Server Side of the Cluster Root

You must notice that the cluster root has a component, peer, the instance of Peer, as defined in Line 52 of List 7.2. If you are familiar with what I discuss in Chapter 5 and Chapter 6, the component should follow the pattern of SD. Yes, you are right. Now let us browse the code of List 7.24. Table 7.7 illustrates the pattern of SD.

```
1    package com.greatfree.cluster.root;
2
3    import java.util.Calendar;
4
5    import com.greatfree.cluster.message.ClusterMessageType;
6    import com.greatfree.cluster.message.HelloWorldAnycastResponse;
7    import com.greatfree.cluster.message.HelloWorldBroadcastResponse;
8    import com.greatfree.cluster.message.HelloWorldUnicastResponse;
9    import com.greatfree.cluster.message.ShutdownChildrenAdminNotification;
10   import com.greatfree.concurrency.NotificationDispatcher;
11   import com.greatfree.message.ServerMessage;
12   import com.greatfree.remote.OutMessageStream;
13   import com.greatfree.server.ServerDispatcher;
14   import com.greatfree.testing.data.ServerConfig;
15
16   /*
17    * The dispatcher is employed by the root of the cluster to dispatch messages from the distributed nodes
18    * within the cluster. 05/10/2017, Bing Li
19    */
20
21   // Created: 05/10/2017, Bing Li
22   public class RootDispatcher extends ServerDispatcher
23   {
24       // Declare a notification dispatcher to receive remote responses which aim to answer the
25       // broadcast request. 05/21/2017, Bing Li
26       private NotificationDispatcher<HelloWorldBroadcastResponse, HelloWorldBroadcastResponseThread,
```

```
27          HelloWorldBroadcastResponseThreadCreator> helloWorldBroadcastResponseDispatcher;
28
29      // Declare a notification dispatcher to receive remote responses which aim to answer the
30      // anycast request. 05/21/2017, Bing Li
31      private NotificationDispatcher<HelloWorldAnycastResponse, HelloWorldAnycastResponseThread,
32          HelloWorldAnycastResponseThreadCreator> helloWorldAnycastResponseDispatcher;
33
34      // Declare a notification dispatcher to receive remote responses which aim to answer
35      // the unicast request. 05/21/2017, Bing Li
36      private NotificationDispatcher<HelloWorldUnicastResponse, HelloWorldUnicastResponseThread,
37          HelloWorldUnicastResponseThreadCreator> helloWorldUnicastResponseDispatcher;
38
39      // Declare a notification dispatcher to process ShutdownChildrenAdminNotification to the child
40      // when such a notification is received. 05/19/2016, Bing Li
41      private NotificationDispatcher<ShutdownChildrenAdminNotification,
42          ShutdownChildrenAdminNotificationThread, ShutdownChildrenAdminNotificationThreadCreator>
43          shutdownChildrenNotificationDispatcher;
44
45      public RootDispatcher(int threadPoolSize, long threadKeepAliveTime, int schedulerPoolSize,
46          long schedulerKeepAliveTime)
47      {
48          super(threadPoolSize, threadKeepAliveTime, schedulerPoolSize, schedulerKeepAliveTime);
49
50          // Initialize the notification dispatcher for the notification,
51          // HelloWorldBroadcastResponse. 11/27/2014, Bing Li
52          this.helloWorldBroadcastResponseDispatcher = new NotificationDispatcher
53              .NotificationDispatcherBuilder<HelloWorldBroadcastResponse,
54              HelloWorldBroadcastResponseThread, HelloWorldBroadcastResponseThreadCreator>()
55              .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
56              .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
57              .threadCreator(new HelloWorldBroadcastResponseThreadCreator())
58              .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
59              .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
60              .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
61              .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
62              .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
63              .scheduler(super.getSchedulerPool())
64              .build();
65
66          // Initialize the notification dispatcher for the notification,
67          // HelloWorldAnycastResponse. 11/27/2014, Bing Li
68          this.helloWorldAnycastResponseDispatcher = new NotificationDispatcher
69              .NotificationDispatcherBuilder<HelloWorldAnycastResponse, HelloWorldAnycastResponseThread,
70              HelloWorldAnycastResponseThreadCreator>()
71              .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
72              .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
73              .threadCreator(new HelloWorldAnycastResponseThreadCreator())
74              .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
75              .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
76              .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
77              .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
78              .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
79              .scheduler(super.getSchedulerPool())
80              .build();
81
82          // Initialize the notification dispatcher for the notification,
83          // HelloWorldUnicastResponse. 11/27/2014, Bing Li
84          this.helloWorldUnicastResponseDispatcher = new NotificationDispatcher
85              .NotificationDispatcherBuilder<HelloWorldUnicastResponse, HelloWorldUnicastResponseThread,
86              HelloWorldUnicastResponseThreadCreator>()
87              .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
88              .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
89              .threadCreator(new HelloWorldUnicastResponseThreadCreator())
90              .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
91              .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
92              .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
93              .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
94              .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
95              .scheduler(super.getSchedulerPool())
96              .build();
97
98          // Initialize the notification dispatcher for the notification,
99          // ShutdownChildrenAdminNotification. 11/27/2014, Bing Li
100         this.shutdownChildrenNotificationDispatcher = new NotificationDispatcher
101             .NotificationDispatcherBuilder<ShutdownChildrenAdminNotification,
102             ShutdownChildrenAdminNotificationThread, ShutdownChildrenAdminNotificationThreadCreator>()
103             .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
```

```
104             .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
105             .threadCreator(new ShutdownChildrenAdminNotificationThreadCreator())
106             .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
107             .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
109             .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
110             .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
111             .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
112             .scheduler(super.getSchedulerPool())
113             .build();
114     }
115
116     /*
117      * Shutdown the dispatcher. 11/25/2014, Bing Li
118      */
119     public void shutdown() throws InterruptedException
120     {
121         this.helloWorldBroadcastResponseDispatcher.dispose();
122         this.helloWorldAnycastResponseDispatcher.dispose();
123         this.helloWorldUnicastResponseDispatcher.dispose();
124         this.shutdownChildrenNotificationDispatcher.dispose();
125         super.shutdown();
126     }
127
128     /*
129      * Process the available messages in a concurrent way. 04/17/2017, Bing Li
130      */
131     public void consume(OutMessageStream<ServerMessage> message)
132     {
133         // Check the types of received messages. 04/17/2017, Bing Li
134         switch (message.getMessage().getType())
135         {
136             case ClusterMessageType.HELLO_WORLD_BROADCAST_RESPONSE:
137                 System.out.println("HELLO_WORLD_BROADCAST_RESPONSE received @" +
138                     Calendar.getInstance().getTime());
139                 // Check whether the shutdown notification dispatcher is ready or not. 02/15/2016, Bing Li
140                 if (!this.helloWorldBroadcastResponseDispatcher.isReady())
141                 {
142                     // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
143                     super.execute(this.helloWorldBroadcastResponseDispatcher);
144                 }
145                 // Enqueue the instance of HelloWorldBroadcastResponse into the dispatcher for
146                 // concurrent processing. 02/15/2016, Bing Li
147                 this.helloWorldBroadcastResponseDispatcher.enqueue((HelloWorldBroadcastResponse)
148                     message.getMessage());
149                 break;
150
151             case ClusterMessageType.HELLO_WORLD_ANYCAST_RESPONSE:
152                 System.out.println("HELLO_WORLD_ANYCAST_RESPONSE received @" +
153                     Calendar.getInstance().getTime());
154                 // Check whether the shutdown notification dispatcher is ready or not. 02/15/2016, Bing Li
155                 if (!this.helloWorldAnycastResponseDispatcher.isReady())
156                 {
157                     // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
158                     super.execute(this.helloWorldAnycastResponseDispatcher);
159                 }
160                 // Enqueue the instance of HelloWorldAnycastResponse into the dispatcher for concurrent
161                 // processing. 02/15/2016, Bing Li
162                 this.helloWorldAnycastResponseDispatcher.enqueue((HelloWorldAnycastResponse)
163                     message.getMessage());
164                 break;
165
166             case ClusterMessageType.HELLO_WORLD_UNICAST_RESPONSE:
167                 System.out.println("HELLO_WORLD_UNICAST_RESPONSE received @" +
168                     Calendar.getInstance().getTime());
169                 // Check whether the shutdown notification dispatcher is ready or not. 02/15/2016, Bing Li
170                 if (!this.helloWorldUnicastResponseDispatcher.isReady())
171                 {
172                     // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
173                     super.execute(this.helloWorldUnicastResponseDispatcher);
174                 }
175                 // Enqueue the instance of HelloWorldUnicastResponse into the dispatcher for concurrent
176                 // processing. 02/15/2016, Bing Li
177                 this.helloWorldUnicastResponseDispatcher.enqueue((HelloWorldUnicastResponse)
178                     message.getMessage());
179                 break;
180
181             case ClusterMessageType.SHUTDOWN_CHILDREN_ADMIN_NOTIFICATION:
```

```
182                 System.out.println("SHUTDOWN_CHILDREN_ADMIN_NOTIFICATION received @" +
183                     Calendar.getInstance().getTime());
184                 // Check whether the shutdown notification dispatcher is ready or not. 02/15/2016, Bing Li
185                 if (!this.shutdownChildrenNotificationDispatcher.isReady())
186                 {
187                     // Execute the notification dispatcher concurrently. 02/15/2016, Bing Li
188                     super.execute(this.shutdownChildrenNotificationDispatcher);
189                 }
190                 // Enqueue the instance of ShutdownChildrenAdminNotification into the dispatcher for
191                 // concurrent processing. 02/15/2016, Bing Li
192                 this.shutdownChildrenNotificationDispatcher.enqueue((ShutdownChildrenAdminNotification)
193                     message.getMessage());
194                 break;
195             }
196         }
197     }
```

List 7.24 The code of RootDispatcher.java

Pattern: SD (Server Dispatcher)		
Class	Peer<RootDispatcher>	
Singleton	No	
Method	void consume(OutMessageStream<ServerMessage> message)	
	void shutdown()	
Enclosed Pattern	ND (Notification Dispatcher)	NotificationDispatcher<HelloWorldBroadcastResponse, HelloWorldBroadcastResponseThread, HelloWorldBroadcastResponseThreadCreator>
		NotificationDispatcher<HelloWorldAnycastResponse, HelloWorldAnycastResponseThread, HelloWorldAnycastResponseThreadCreator>
		NotificationDispatcher<HelloWorldUnicastResponse, HelloWorldUnicastResponseThread, HelloWorldUnicastResponseThreadCreator>
		NotificationDispatcher<ShutdownChildrenAdminNotification, ShutdownChildrenAdminNotificationThread, ShutdownChildrenAdminNotificationThreadCreator>
Employed API	ServerDispatcher	
	NotificationDispatcher<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>, ThreadCreator extends NotificationThreadCreatable<Notification, NotificationThread>>	

Table 7.15 The pattern of SD for the cluster root

In accordance with Table 7.7, you must have the same consensus as me again. That is, the code for the server side is always highly patterned for developers to program comfortably with simplified behaviors, CPR.

You might have a question why we should have a SD on the server side or which nodes send those messages to the root. In our case, only two types distributed nodes are available, i.e., the root and the children. Thus, the messages received by the root are originated from the children for sure. So the next question is what are the scenarios that those children need to sent messages to the root. The only case that a child needs to send a message to the root in our system is that the child receives a broadcast request during one procedure of request multicasting. After that, the child is required to deal with the request and generate the corresponding response. When the response is available, it should be sent back to the root. Therefore, you can take a look at the NDs enclosed in the SD, as shown in List 7.23 and Table 7.7. In the previous code, we have read many code in the pattern of ND, such as Section 2.6 of Chapter 5, Section 2.7 and Section 3.5 of Chapter 6. All of the NDs have messages which are named as notifications. However, in our current case, the ND has to process messages which are named as responses, such HelloWorldBroadcastResponse, HelloWorldAnycastResponse and HelloWorldUnicastResponse since all of them are generated by children to respond broadcasting requests. Well, from the syntax's point

of view, it is reasonable because all of messages are derived from ServerMessage in GreatFree.

So let me give you simple explanations to the code of RootDispatcher.java in List 7.3.

Line 26~27 declares an instance, helloWorldBroadcastResponseDispatcher, of NotificationDispatcher (ND). It is responsible for dispatching received messages in the type of HelloWorldBroadcastResponse.

Line 31~32 declares an instance, helloWorldAnycastResponseDispatcher, of NotificationDispatcher (ND). It dispatches received messages of HelloWorldAnycastResponse to its internal managed notification queues.

Line 36~37 declares an instance, helloWorldUnicastResponseDispatcher, of NotificationDispatcher (ND). It needs to dispatch responses from children to its built-in notifications queues.

Line 41~43 declares an instance, shutdownChildrenNotificationDispatcher, of NotificationDispatcher (ND). Its form is exactly identical to classic NDs we discuss previously in the sense that the messages it processes is named as notifications.

Line 45~114 initializes the NDs.

Line 119~126 defines the method, dispose(), to collect resources in the SD.

Line 131~196 defines the method, consume(), to dispatch received messages concurrently.

2.6 ND: Accomplishing Request Multicasting

Although ND is not a new stuff to you, I have to spend a little more time on the NDs since they are employed in multicasting. In essence, there is no any difference. However, it is necessary to emphasize that the NDs in multicasting are the indispensible components to accomplishing request multicasting.

Pattern: ND (Notification Dispatcher)		
Class	NotificationDispatcher<HelloWorldBroadcastResponse, HelloWorldBroadcastResponseThread, HelloWorldBroadcastResponseThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(HelloWorldBroadcastResponse notification)	
Enclosed Pattern	Notification	HelloWorldBroadcastResponse
	NQ (Notification Queue)	HelloWorldBroadcastResponseThread
	NQC (Notification Queue Creator)	HelloWorldBroadcastResponseThreadCreator
Employed API	ServerMessage	
	NotificationQueue<Notification extends ServerMessage>	
	NotificationThreadCreatable<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>>	

Table 7.16 The pattern of ND for HelloWorldBroadcastResponse

2.6.1 ND for HelloWorldBroadcastResponse

When a broadcast request is received by each child in the cluster, those children need to generate corresponding responses with respect to the request. After the responses are created, they are sent back to the cluster root by the child. Until each child's response is received, the requesting multicasting is done. It is possible that some children cannot send their responses to the root for some possible problems, such as networks or other faults. So, the root should not always wait for children's responses. It is necessary to set a timeout for the waiting. Table 7.16 summarizes the pattern of ND for HelloWorldBroadcastResponse.

Let us check the code. List 7.16 shows the notification, HelloWorldBroadcastResponse, for the ND.

The notification queue, HelloWorldBroadcastResponseThread.java, is shown in List 7.25. Once if the response is received, it invokes the method, ClusterRoot.CLUSTER().broadcastResponseReceived(), to collect the responses.

```
1    package com.greatfree.cluster.root;
2
3    import com.greatfree.cluster.message.HelloWorldBroadcastResponse;
4    import com.greatfree.concurrency.NotificationQueue;
5    import com.greatfree.testing.data.ServerConfig;
6
7    /*
8     * The thread receives the remote responses that answer the broadcast request
9     * concurrently. 05/21/2017, Bing Li
10    */
11
12   // Created: 05/21/2017, Bing Li
13   public class HelloWorldBroadcastResponseThread extends NotificationQueue
14       <HelloWorldBroadcastResponse>
15   {
16       public HelloWorldBroadcastResponseThread(int taskSize)
17       {
18           super(taskSize);
19       }
20
21       @Override
22       public void run()
23       {
24           HelloWorldBroadcastResponse response;
25           while (!this.isShutdown())
26           {
27               while (!this.isEmpty())
28               {
29                   try
30                   {
31                       response = this.getNotification();
32                       // Collect responses from the children. 05/21/2017, Bing Li
33                       ClusterRoot.CLUSTER().broadcastResponseReceived(response);
34                       this.disposeMessage(response);
35                   }
36                   catch (InterruptedException e)
37                   {
38                       e.printStackTrace();
39                   }
40               }
41               try
42               {
43                   // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
44                   this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
45               }
46               catch (InterruptedException e)
47               {
48                   e.printStackTrace();
49               }
```

```
50            }
51        }
52    }
```

List 7.25 The code of HelloWorldBroadcastResponseThread.java

To be general, the thread must follow the pattern of DWC, as presented in Table 7.17.

Pattern: DWC (Double While Concurrency)		
Class	HelloWorldBroadcastResponseThread	
Singleton	No	
Method	void dispose()	
	void enqueue(HelloWorldBroadcastResponse notification)	
	HelloWorldBroadcastResponse getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	HelloWorldBroadcastResponse
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 7.17 The pattern of DWC to collect received responses, HelloWorldBroadcastResponse, concurrently for request broadcasting

The creator of the notification queue is shown in List 7.25.

```
1     package com.greatfree.cluster.root;
2
3     import com.greatfree.cluster.message.HelloWorldBroadcastResponse;
4     import com.greatfree.concurrency.NotificationThreadCreatable;
5
6     // Created: 05/21/2017, Bing Li
7     public class HelloWorldBroadcastResponseThreadCreator implements NotificationThreadCreatable
8         <HelloWorldBroadcastResponse, HelloWorldBroadcastResponseThread>
9     {
10        @Override
11        public HelloWorldBroadcastResponseThread createNotificationThreadInstance(int taskSize)
12        {
13            return new HelloWorldBroadcastResponseThread(taskSize);
14        }
15    }
```

List 7.26 The code of HelloWorldBroadcastResponseThreadCreator.java

2.6.2 ND for HelloWorldAnycastResponse

The ND for HelloWorldAnycastResponse is summarized in Table 7.18.

Pattern: ND (Notification Dispatcher)		
Class	NotificationDispatcher<HelloWorldAnycastResponse, HelloWorldAnycastResponseThread, HelloWorldAnycastResponseThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(HelloWorldAnycastResponse notification)	
Enclosed Pattern	Notification	HelloWorldAnycastResponse
	NQ (Notification Queue)	HelloWorldAnycastResponseThread
	NQC (Notification Queue Creator)	HelloWorldAnycastResponseThreadCreator
Employed API	ServerMessage	
	NotificationQueue<Notification extends ServerMessage>	
	NotificationThreadCreatable<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>>	

Table 7.18 The pattern of ND to dispatch the notification, HelloWorldAnycastResponse

The notification, HelloWorldAnycastResponse, is shown in List 7.19.

The notification queue, HelloWorldAnycastResponseThread, is shown in List 7.27. It invokes the method, anycastResponseReceived(), at Line 32 to collect the responses.

```
1   package com.greatfree.cluster.root;
2
3   import com.greatfree.cluster.message.HelloWorldAnycastResponse;
4   import com.greatfree.concurrency.NotificationQueue;
5   import com.greatfree.testing.data.ServerConfig;
6
7   /*
8    * The thread receives the remote responses that answer the anycast request concurrently. 05/21/2017, Bing Li
9    */
10
11  // Created: 05/21/2017, Bing Li
12  public class HelloWorldAnycastResponseThread extends NotificationQueue
13      <HelloWorldAnycastResponse>
14  {
15      public HelloWorldAnycastResponseThread(int taskSize)
16      {
17          super(taskSize);
18      }
19
20      @Override
21      public void run()
22      {
23          HelloWorldAnycastResponse response;
24          while (!this.isShutdown())
25          {
26              while (!this.isEmpty())
27              {
28                  try
29                  {
30                      response = this.getNotification();
31                      // Collect responses from the children. 05/21/2017, Bing Li
32                      ClusterRoot.CLUSTER().anycastResponseReceived(response);
33                      this.disposeMessage(response);
34                  }
35                  catch (InterruptedException e)
36                  {
37                      e.printStackTrace();
38                  }
39              }
40              try
41              {
42                  // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
43                  this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
44              }
45              catch (InterruptedException e)
46              {
47                  e.printStackTrace();
48              }
49          }
50      }
51  }
```

List 7.27 The code of HelloWorldAnycastResponseThread.java

Table 7.19 presents the pattern of DWC to process the received response, HelloWorldAnycastResponse.

The last component for the ND is the notification creator. The code is shown in List 7.28. In the previous chapters, only the messages named as notifications are placed with NotificationThreadCreatable together. However, now you do that with the message, HelloWorldAnycastResponse. Anyway, the so-called response has the same parent, ServerMessage, as those notifications. So it is absolutely valid.

```
1    package com.greatfree.cluster.root;
2
3    import com.greatfree.cluster.message.HelloWorldAnycastResponse;
4    import com.greatfree.concurrency.NotificationThreadCreatable;
5
6    // Created: 05/21/2017, Bing Li
7    public class HelloWorldAnycastResponseThreadCreator implements NotificationThreadCreatable
8        <HelloWorldAnycastResponse, HelloWorldAnycastResponseThread>
9    {
10       @Override
11       public HelloWorldAnycastResponseThread createNotificationThreadInstance(int taskSize)
12       {
13           return new HelloWorldAnycastResponseThread(taskSize);
14       }
15   }
```

List 7.28 The code of HelloWorldAnycastResponseThreadCreator.java

Pattern: DWC (Double While Concurrency)		
Class	HelloWorldAnycastResponseThread	
Singleton	No	
Method	void dispose()	
	void enqueue(HelloWorldAnycastResponse notification)	
	HelloWorldAnycastResponse getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	HelloWorldAnycastResponse
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 7.19 The pattern of DWC to collect the received response, HelloWorldAnycastResponse, concurrently for request anycasting

2.6.3 ND for HelloWorldUnicastResponse

Let us show the pattern table for the ND to you first, as shown in Table 7.20.

Pattern: ND (Notification Dispatcher)		
Class	NotificationDispatcher<HelloWorldUnicastResponse, HelloWorldUnicastResponseThread, HelloWorldUnicastResponseThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(HelloWorldUnicastResponse notification)	
Enclosed Pattern	Notification	HelloWorldUnicastResponse
	NQ (Notification Queue)	HelloWorldUnicastResponseThread
	NQC (Notification Queue Creator)	HelloWorldUnicastResponseThreadCreator
Employed API	ServerMessage	
	NotificationQueue<Notification extends ServerMessage>	
	NotificationThreadCreatable<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>>	

Table 7.20 The pattern of ND to dispatch the notification, HelloWorldUnicastResponse

The notification, HelloWorldUnicastResponse, has already shown you in List 7.21.

The notification queue, i.e., the relevant thread, has to be programmed by you. List 7.29 presents the code of HelloWorldUnicastResponseThread.java. The thread follows the pattern of DWC as shown in Table 7.21. Well, it invokes the method, unicastResponseReceived(), to collect the responses. For request unicasting, only one response is received at most.

```
1   package com.greatfree.cluster.root;
2
3   import com.greatfree.cluster.message.HelloWorldUnicastResponse;
4   import com.greatfree.concurrency.NotificationQueue;
5   import com.greatfree.testing.data.ServerConfig;
6
7   /*
8    * The thread receives the remote responses that answer the unicast request concurrently. 05/21/2017, Bing Li
9    */
10
11  // Created: 05/21/2017, Bing Li
12  public class HelloWorldUnicastResponseThread extends NotificationQueue<HelloWorldUnicastResponse>
13  {
14      public HelloWorldUnicastResponseThread(int taskSize)
15      {
16          super(taskSize);
17      }
18
19      @Override
20      public void run()
21      {
22          HelloWorldUnicastResponse response;
23          while (!this.isShutdown())
24          {
25              while (!this.isEmpty())
26              {
27                  try
28                  {
29                      response = this.getNotification();
30                      // Collect responses from the children. 05/21/2017, Bing Li
31                      ClusterRoot.CLUSTER().unicastResponseReceived(response);
32                      this.disposeMessage(response);
33                  }
34                  catch (InterruptedException e)
35                  {
36                      e.printStackTrace();
37                  }
38              }
39              try
40              {
41                  // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
42                  this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
43              }
44              catch (InterruptedException e)
45              {
46                  e.printStackTrace();
47              }
48          }
49      }
50  }
```

List 7.29 The code of HelloWorldUnicastResponseThread.java

Pattern: DWC (Double While Concurrency)		
Class	HelloWorldUnicastResponseThread	
Singleton	No	
Method	void dispose()	
	void enqueue(HelloWorldUnicastResponse notification)	
	HelloWorldUnicastResponse getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	HelloWorldUnicastResponse
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 7.21 The pattern of DWC to collect received responses, HelloWorldUnicastResponse, concurrently for request unicasting

The creator of the notification, HelloWorldUnicastResponse, is named as HelloWorldUnicastResponseCreator in our convention. Its code is presented in List 7.30.

```
1    package com.greatfree.cluster.root;
2
3    import com.greatfree.cluster.message.HelloWorldUnicastResponse;
4    import com.greatfree.concurrency.NotificationThreadCreatable;
5
6    // Created: 05/21/2017, Bing Li
7    public class HelloWorldUnicastResponseThreadCreator implements NotificationThreadCreatable
8        <HelloWorldUnicastResponse, HelloWorldUnicastResponseThread>
9    {
10       @Override
11       public HelloWorldUnicastResponseThread createNotificationThreadInstance(int taskSize)
12       {
13           return new HelloWorldUnicastResponseThread(taskSize);
14       }
15   }
```

List 7.30 The code of HelloWorldUnicastResponseCreator.java

2.6.4 ND for ShutdownChildrenAdminNotification

The last ND declared in the SD for the cluster root is the one named as shutdownChildrenNotificationDispatcher at Line 41 of RootDispatcher.java in List 7.23. Even without my further explanations, you should be able to handle it for sure since we have seen such typical ND for many times in Chapter 5 and Chapter 6.

The ND does not serve for any request multicasting as its brothers do in Section 2.6.1, Section 2.6.2 and Section 2.6.3. It receives an administration instruction from one administrator such that the root is forced to send a broadcast notification to each of children within the cluster to shut down. It happens when the system should be terminated. Table 7.22 illustrates the ND.

Pattern: ND (Notification Dispatcher)		
Class	NotificationDispatcher<ShutdownChildrenAdminNotification, ShutdownChildrenAdminNotificationThread, ShutdownChildrenAdminNotificationThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(ShutdownChildrenAdminNotification notification)	
Enclosed Pattern	Notification	ShutdownChildrenAdminNotification
	NQ (Notification Queue)	ShutdownChildrenAdminNotificationThread
	NQC (Notification Queue Creator)	ShutdownChildrenAdminNotificationThreadCreator
Employed API	ServerMessage	
	NotificationQueue<Notification extends ServerMessage>	
	NotificationThreadCreatable<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>>	

Table 7.22 The pattern of ND to dispatch the administration instruction, ShutdownChildrenAdminNotification

The notification, ShutdownChildrenAdminNotification, is exhibited in List 7.31.

```
1    package com.greatfree.cluster.message;
2
3    import com.greatfree.message.ServerMessage;
4
5    /*
6     * The notification works as a remote command from the administrator to the root of the cluster.
```

```
7      * Then, the root broadcasts it to
8      */
9
10     // Created: 05/19/2017, Bing Li
11     public class ShutdownChildrenAdminNotification extends ServerMessage
12     {
13         private static final long serialVersionUID = -5803616520557522304L;
14
15         public ShutdownChildrenAdminNotification()
16         {
17             super(ClusterMessageType.SHUTDOWN_CHILDREN_ADMIN_NOTIFICATION);
18         }
19     }
```

List 7.31 The code of ShutdownChildrenAdminNotification.java

The notification queue, ShutdownChildrenAdminNotificationThread, is presented in List 7.32 in the form of DWC. When it receives a remote notification, ShutdownChildrenAdminNotification, it invokes the method, broadcastShutdownNotify(), of ClusterRoot (List 7.2) at Line 31~32 of List 7.32. The pattern of the ND is exhibited in Table 7.23.

You must remember we talk about the issue of transforming a blocking method to a non-blocking one indirectly in Section 2.2, right? Here is the example how to do that. The method, broadcastShutdownNotify(), of ClusterRoot (List 7.2) is invoked by an instance of NotificationQueue. Thus, the blocking method is turned into a non-blocking one implicitly although we do not do that explicitly.

```
1      package com.greatfree.cluster.root;
2
3      import java.io.IOException;
4
5      import com.greatfree.cluster.ClusterConfig;
6      import com.greatfree.cluster.message.ShutdownChildrenAdminNotification;
7      import com.greatfree.concurrency.NotificationQueue;
8      import com.greatfree.testing.data.ServerConfig;
9
10     // Created: 05/19/2017, Bing Li
11     public class ShutdownChildrenAdminNotificationThread extends NotificationQueue
12         <ShutdownChildrenAdminNotification>
13     {
14         public ShutdownChildrenAdminNotificationThread(int taskSize)
15         {
16             super(taskSize);
17         }
18
19         @Override
20         public void run()
21         {
22             ShutdownChildrenAdminNotification notification;
23             while (!this.isShutdown())
24             {
25                 while (!this.isEmpty())
26                 {
27                     try
28                     {
29                         notification = this.getNotification();
30                         // Broadcast shutdown notification to all of its children. 05/20/2017, Bing Li
31                         ClusterRoot.CLUSTER().broadcastShutdownNotify
32                             (ClusterConfig.ROOT_BRANCH_COUNT, ClusterConfig.SUB_BRANCH_COUNT);
33                         this.disposeMessage(notification);
34                     }
35                     catch (InterruptedException | InstantiationException | IllegalAccessException | IOException e)
36                     {
37                         e.printStackTrace();
38                     }
39                 }
40                 try
41                 {
```

```
42                  // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
43                  this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
44              }
45              catch (InterruptedException e)
46              {
47                  e.printStackTrace();
48              }
49          }
50      }
51  }
```

List 7.32 The code of ShutdownChildrenAdminNotificationThread.java

Pattern: DWC (Double While Concurrency)		
Class	ShutdownChildrenAdminNotificationThread	
Singleton	No	
Method	void dispose()	
	void enqueue(ShutdownChildrenAdminNotification notification)	
	ShutdownChildrenAdminNotification getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	ShutdownChildrenAdminNotification
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 7.23 The pattern of ND to invoke the broadcast notification to shut down all of children when receiving the notification, ShutdownChildrenAdminNotification

The last component for the ND is named as ShutdownChildrenAdminNotificationThreadCreator according to our convention. Its code is exhibited in List 7.33.

```
1   package com.greatfree.cluster.root;
2
3   import com.greatfree.cluster.message.ShutdownChildrenAdminNotification;
4   import com.greatfree.concurrency.NotificationThreadCreatable;
5
6   // Created: 05/20/2017, Bing Li
7   public class ShutdownChildrenAdminNotificationThreadCreator implements NotificationThreadCreatable
8       <ShutdownChildrenAdminNotification, ShutdownChildrenAdminNotificationThread>
9   {
10      @Override
11      public ShutdownChildrenAdminNotificationThread createNotificationThreadInstance(int taskSize)
12      {
13          return new ShutdownChildrenAdminNotificationThread(taskSize);
14      }
15  }
```

List 7.33 The code of ShutdownChildrenAdminNotificationThreadCreator.java

2.7 A Summary

Now all of the code for the cluster root is accomplished. You should have encountered a new challenge during the procedure since many new stuffs have to be learned. Fortunately, you should feel comfortable when programming such a difficult distributed node with the support of GreatFree APIs and patterns. Your previous experiences of CPR still work effectively with the scale of the system is becoming much larger. You know you are implementing a huge system with your distributed knowledge. Meanwhile, your workload is low enough to get it done in one afternoon.

3. Programming the Child

It is time for us to program the child of the cluster. The difference between the root and the child is distinctive in terms of their contributions to the cluster. First of all, all of computations within a cluster are originated from the root since any tasks can be sent to the children within the cluster only by the root through multicasting. In addition, the number of the root is one whereas that of the cluster is infinite. The root has to organize those children into one particular hierarchical structure to start any computing. On the other hand, it is possible that a child is able to play the role to send the original message. That is, one child in the cluster sends the initial message to the root and then the root forwards it to the children through multicasting. Anyway, it is convenient to achieve any of the application cases with GreatFree. Let us now keep working on the contrived system to construct the cluster using those multicasting APIs and patterns.

3.1 SP: Starting Up the Child

When programming any distributed applications, the initial question is almost always like this, "what is the distributed component?". We have answered the question at the beginning of the chapter. The child we attempt to program should be implemented with the API of Peer. Now let us start up the child first. Its code is exhibited in List 7.34. For it follows the pattern of SP, Table 7.24 illustrates the pattern clearly.

```
1    package com.greatfree.cluster.child;
2
3    import java.io.IOException;
4
5    import com.greatfree.exceptions.RemoteReadException;
6    import com.greatfree.testing.data.ServerConfig;
7    import com.greatfree.util.TerminateSignal;
8
9    // Created: 05/12/2017, Bing Li
10   public class StartChild
11   {
12       public static void main(String[] args)
13       {
14           System.out.println("Starting up the child ...");
15           try
16           {
17               ClusterChild.CLUSTER().start();
18           }
19           catch (ClassNotFoundException | IOException | RemoteReadException e)
20           {
21               e.printStackTrace();
22           }
23           System.out.println("The child is started ...");
24
25           // After the server is started, the loop check whether the flag of terminating is set. If the terminating flag
26           // is true, the process is ended. Otherwise, the process keeps running. 08/22/2014, Bing Li
27           while (!TerminateSignal.SIGNAL().isTerminated())
28           {
29               try
30               {
31                   // If the terminating flag is false, it is required to sleep for some time. Otherwise, it might
32                   // cause the high CPU usage. 08/22/2014, Bing Li
33                   Thread.sleep(ServerConfig.TERMINATE_SLEEP);
34               }
35               catch (InterruptedException e)
36               {
37                   e.printStackTrace();
38               }
39           }
40       }
```

```
41    }
```

List 7.34 The code of StartChild.java

Pattern: SP (Starting Point)		
Class	StartChild	
Singleton	Yes	
Method	void main(String[] args)	
Enclosed Pattern	MS (Main Server)	ClusterChild
Employed API	TerminalSignal	

Table 7.24 The pattern of SP to start up the child

3.2 CC: Programming the Client Side of the Cluster Child

You must not have any knowledge questions about the cluster as a distribute system. So we just spend most of our time on the code. We see the new code at Line 17 of StartChild.jave in List 7.34. According to our experiences, that is the most important one since it has such a counterpart named ClusterRoot as shown in List 7.2. The code of ClusterChild.java is presented in List 7.35.

```
1     package com.greatfree.cluster.child;
2
3     import java.io.IOException;
4
5     import com.greatfree.chat.ChatConfig;
6     import com.greatfree.chat.message.peer.ChatRegistryRequest;
7     import com.greatfree.cluster.ClusterConfig;
8     import com.greatfree.cluster.message.HelloWorldBroadcastNotification;
9     import com.greatfree.cluster.message.HelloWorldBroadcastRequest;
10    import com.greatfree.cluster.message.RootIPAddressBroadcastNotification;
11    import com.greatfree.cluster.message.ShutdownChildrenBroadcastNotification;
12    import com.greatfree.cluster.message.child.HelloWorldBroadcastNotificationCreator;
13    import com.greatfree.cluster.message.child.HelloWorldBroadcastRequestCreator;
14    import com.greatfree.cluster.message.child.RootIPAddressBroadcastNotificationCreator;
15    import com.greatfree.cluster.message.child.ShutdownChildrenBroadcastNotificationCreator;
16    import com.greatfree.clustering.child.ClusterChildBroadcastNotifier;
17    import com.greatfree.clustering.child.ClusterChildBroadcastReader;
18    import com.greatfree.exceptions.RemoteReadException;
19    import com.greatfree.message.ServerMessage;
20    import com.greatfree.server.Peer;
21    import com.greatfree.testing.data.ServerConfig;
22    import com.greatfree.util.IPAddress;
23    import com.greatfree.util.TerminateSignal;
24    import com.greatfree.util.Tools;
25    import com.greatfree.util.UtilConfig;
26
27    /*
28     * The distributed node within one particular cluster. 05/11/2017, Bing Li
29     */
30
31    // Created: 05/11/2017, Bing Li
32    public class ClusterChild
33    {
34        // The child peer in the cluster. 05/12/2017, Bing Li
35        private Peer<ChildDispatcher> childPeer;
36        // The IP address of the cluster root. 06/15/2017, Bing Li
37        private IPAddress rootAddress;
38
39        // The root IP address should be broadcast to each node in the cluster before broadcasting
40        // requesting/responding can be performed. 05/20/2017, Bing Li
41        private ClusterChildBroadcastNotifier<RootIPAddressBroadcastNotification,
42            RootIPAddressBroadcastNotificationCreator> rootIPBroadcastNotifier;
43
44        // The child broadcastor for the notification of HelloWorldBroadcastNotification. 05/12/2017, Bing Li
45        private ClusterChildBroadcastNotifier<HelloWorldBroadcastNotification,
46            HelloWorldBroadcastNotificationCreator> helloWorldBroadcastNotifier;
47
```

```
48      // The child broadcastor for the request of HelloWorldBroadcastRequest. 05/20/2017, Bing Li
49      private ClusterChildBroadcastReader<HelloWorldBroadcastRequest,
50          HelloWorldBroadcastRequestCreator> helloWorldBroadcastReader;
51
52      // The child broadcastor for the notification of ShutdownChildrenBroadcastNotification. 05/19/2017, Bing Li
53      private ClusterChildBroadcastNotifier<ShutdownChildrenBroadcastNotification,
54          ShutdownChildrenBroadcastNotificationCreator> shutdownBroadcastNotifier;
55
56      private ClusterChild()
57      {
58      }
59
60      private static ClusterChild instance = new ClusterChild();
61
62      public static ClusterChild CLUSTER()
63      {
64          if (instance == null)
65          {
66              instance = new ClusterChild();
67              return instance;
68          }
69          else
70          {
71              return instance;
72          }
73      }
74
75      /*
76       * Stop the descendant peer. 05/12/2017, Bing Li
77       */
78      public void stop() throws IOException, InterruptedException, ClassNotFoundException,
79          RemoteReadException
80      {
81          // Set the terminating signal. 11/25/2014, Bing Li
82          TerminateSignal.SIGNAL().setTerminated();
83          // Stop the descendant peer. 04/30/2017, Bing Li
84          this.childPeer.stop();
85          // Dispose the root IP notifier. 05/20/2017, Bing Li
86          this.rootIPBroadcastNotifier.dispose();
87          // Dispose the hello world notifier. 05/15/2017, Bing Li
88          this.helloWorldBroadcastNotifier.dispose();
89          // Dispose the hello world reader. 05/20/2017, Bing Li
90          this.helloWorldBroadcastReader.dispose();
91          // Dispose the shutdown notifier. 05/19/2017, Bing Li
92          this.shutdownBroadcastNotifier.dispose();
93      }
94
95      /*
96       * Start the descendant peer. 05/12/2017, Bing Li
97       */
98      public void start() throws IOException, ClassNotFoundException, RemoteReadException
99      {
100         // Initialize the descendant peer. 05/12/2017, Bing Li
101         this.childPeer = new Peer.PeerBuilder<ChildDispatcher>()
102             .peerPort(ChatConfig.CHAT_SERVER_PORT)
103             .peerName(Tools.generateUniqueKey())
104             .sysRegistryServerIP(ChatConfig.CHAT_REGISTRY_ADDRESS)
105             .sysRegistryServerPort(UtilConfig.PEER_REGISTRY_PORT)
106             .listenerCount(ServerConfig.LISTENING_THREAD_COUNT)
107             .listenerThreadPoolSize(ServerConfig.SHARED_THREAD_POOL_SIZE)
108             .listenerThreadKeepAliveTime
109                 (ServerConfig.SHARED_THREAD_POOL_KEEP_ALIVE_TIME)
110             .dispatcher(new ChildDispatcher(ChatConfig.DISPATCHER_THREAD_POOL_SIZE,
111                 ChatConfig.DISPATCHER_THREAD_POOL_KEEP_ALIVE_TIME,
112                 ChatConfig.SCHEDULER_THREAD_POOL_SIZE,
113                 ChatConfig.SCHEDULER_THREAD_POOL_KEEP_ALIVE_TIME))
114             .clientPoolSize(ChatConfig.CLIENT_POOL_SIZE)
115             .readerClientSize(ChatConfig.READER_CLIENT_SIZE)
116             .syncEventerIdleCheckDelay(ChatConfig.SYNC_EVENTER_IDLE_CHECK_DELAY)
117             .syncEventerIdleCheckPeriod(ChatConfig.SYNC_EVENTER_IDLE_CHECK_PERIOD)
118             .syncEventerMaxIdleTime(ChatConfig.SYNC_EVENTER_MAX_IDLE_TIME)
119             .asyncEventQueueSize(ChatConfig.ASYNC_EVENT_QUEUE_SIZE)
120             .asyncEventerSize(ChatConfig.ASYNC_EVENTER_SIZE)
121             .asyncEventingWaitTime(ChatConfig.ASYNC_EVENTING_WAIT_TIME)
122             .asyncEventerWaitTime(ChatConfig.ASYNC_EVENTER_WAIT_TIME)
123             .asyncEventerWaitRound(ChatConfig.ASYNC_EVENTER_WAIT_ROUND)
124             .asyncEventIdleCheckDelay(ChatConfig.ASYNC_EVENT_IDLE_CHECK_DELAY)
```

```
125             .asyncEventIdleCheckPeriod(ChatConfig.ASYNC_EVENT_IDLE_CHECK_PERIOD)
126             .clientThreadPoolSize(ChatConfig.CLIENT_THREAD_POOL_SIZE)
127             .clientThreadKeepAliveTime(ChatConfig.CLIENT_THREAD_KEEP_ALIVE_TIME)
128             .schedulerPoolSize(ChatConfig.SCHEDULER_THREAD_POOL_SIZE)
129             .schedulerKeepAliveTime(ChatConfig.SCHEDULER_THREAD_POOL_KEEP_ALIVE_TIME)
130             .build();
131
132         // Start the descendant peer. 05/12/2017, Bing Li
133         this.childPeer.start();
134
135         // Initialize the root IP broadcast notifier to forward the root IP address to its children. 06/15/2017, Bing Li
136         this.rootIPBroadcastNotifier = new ClusterChildBroadcastNotifier
137             <RootIPAddressBroadcastNotification, RootIPAddressBroadcastNotificationCreator>
138             (this.childPeer.getLocalIPKey(), this.childPeer.getClientPool(),
139             ClusterConfig.MULTICASTOR_POOL_SIZE,
140             ClusterConfig.RESOURCE_WAIT_TIME, new RootIPAddressBroadcastNotificationCreator());
141
142         // Initialize the hello world broadcast notifier. 05/12/2017, Bing Li
143         this.helloWorldBroadcastNotifier = new ClusterChildBroadcastNotifier
144             <HelloWorldBroadcastNotification, HelloWorldBroadcastNotificationCreator>
145             (this.childPeer.getLocalIPKey(), this.childPeer.getClientPool(),
146             ClusterConfig.MULTICASTOR_POOL_SIZE, ClusterConfig.RESOURCE_WAIT_TIME,
147             new HelloWorldBroadcastNotificationCreator());
148
149         // Initialize the hello world broadcast reader. 05/20/2017, Bing Li
150         this.helloWorldBroadcastReader = new ClusterChildBroadcastReader<HelloWorldBroadcastRequest,
151             HelloWorldBroadcastRequestCreator>(this.childPeer.getLocalIPKey(),
152             this.childPeer.getClientPool(), ClusterConfig.MULTICASTOR_POOL_SIZE,
153             ClusterConfig.RESOURCE_WAIT_TIME, new HelloWorldBroadcastRequestCreator());
154
155         // Initialize the shutdown broadcast notifier. 05/19/2017, Bing Li
157         this.shutdownBroadcastNotifier = new ClusterChildBroadcastNotifier
158             <ShutdownChildrenBroadcastNotification, ShutdownChildrenBroadcastNotificationCreator>
159             (this.childPeer.getLocalIPKey(), this.childPeer.getClientPool(),
160             ClusterConfig.MULTICASTOR_POOL_SIZE, ClusterConfig.RESOURCE_WAIT_TIME,
161             new ShutdownChildrenBroadcastNotificationCreator());
162
163         // Register the peer to the chatting registry. 06/15/2017, Bing Li
164         this.childPeer.read(ChatConfig.CHAT_REGISTRY_ADDRESS,
165             ChatConfig.CHAT_REGISTRY_PORT, new ChatRegistryRequest(this.childPeer.getID(),
166             UtilConfig.EMPTY_STRING, UtilConfig.EMPTY_STRING, UtilConfig.EMPTY_STRING));
167     }
168
169     /*
170      * Keep the root IP address. 05/20/2017, Bing Li
171      */
172     public void setRootIP(IPAddress rootAddress)
173     {
174         this.rootAddress = rootAddress;
175     }
176
177     /*
178      * Expose the root IP. 05/20/2017, Bing Li
179      */
180     public String getRootIP()
181     {
182         return this.rootAddress.getIP();
183     }
184
185     /*
186      * Expose the root port. 05/20/2017, Bing Li
187      */
188     public int getRootPort()
189     {
190         return this.rootAddress.getPort();
191     }
192
193     /*
194      * Expose the child's IP. 06/17/2017, Bing Li
195      */
196     public String getChildIP()
197     {
198         return this.childPeer.getPeerIP();
199     }
200
201     /*
202      * Expose the child's port. 06/17/2017, Bing Li
```

```
203         */
204         public int getChildPort()
205         {
206             return this.childPeer.getPort();
207         }
208
209         /*
210          * Respond the response to the root of the cluster. 05/20/2017, Bing Li
211          */
212         public void respondToRoot(ServerMessage response) throws IOException
213         {
214             this.childPeer.syncNotify(this.rootAddress.getIP(), this.rootAddress.getPort(), response);
215         }
216
217         /*
218          * Broadcast the root IP address notification. 05/19/2017, Bing Li
219          */
220         public void broadcastRootIP(RootIPAddressBroadcastNotification notification, int subBranchCount)
221             throws InstantiationException, IllegalAccessException, IOException, InterruptedException
222         {
223             this.rootIPBroadcastNotifier.notifiy(notification, subBranchCount);
224         }
225
226         /*
227          * Broadcast the hello world notification. 05/19/2017, Bing Li
228          */
229         public void broadcastNotify(HelloWorldBroadcastNotification notification, int subBranchCount)
230             throws InstantiationException, IllegalAccessException, IOException, InterruptedException
231         {
232             this.helloWorldBroadcastNotifier.notifiy(notification, subBranchCount);
233         }
234
235         /*
236          * Broadcast the shutdown request. 05/20/2017, Bing Li
237          */
238         public void broadcastRead(HelloWorldBroadcastRequest request, int subBranchCount) throws
239             InstantiationException, IllegalAccessException, IOException, InterruptedException
240         {
241             this.helloWorldBroadcastReader.read(request, subBranchCount);
242         }
243
244         /*
245          * Broadcast the shutdown notification. 05/19/2017, Bing Li
246          */
247         public void broadcastShutdownNotify(ShutdownChildrenBroadcastNotification notification, int
248             subBranchCount) throws InstantiationException, IllegalAccessException, IOException,
249             InterruptedException
250         {
251             this.shutdownBroadcastNotifier.notifiy(notification, subBranchCount);
252         }
253     }
```

List 7.35 The code of ClusterChild.java

You must have the incentive to compare the code of ClusterChild.java (List 7.35) with the one of ClusterRoot.java (List 7.2). What is your image about them? From my point of view, their structures looks similar as well as the code does. However, I do not want to say they share the same pattern. In fact, while their structures are identical, the components within the structures are different in essence. The primary difference originates from the multicasting APIs. In ClusterRoot, the APIs are identified as the root ones whereas those in ClusterChild totally serve multicasting as children. If the behavior of the root is called publishing, that of the child is called forwarding. Additionally, the root, i.e., ClusterRoot, has some APIs that never show up in the child, i.e., ClusterChild, such as the ones for anycasting and unicasting. Why? Moreover, the phenomenon happens to some methods, such as broadcastResponseReceived() (Line 290~293 of List 7.2), anycastResponseReceived() (Line 308~311 of List 7.2) or unicastResponseReceived()

(Line 326~329 of List 7.2). Finally, one method of the child does not exist in the root either, such as respondToRoot() (Line 212~215 of List 7.35).

All of the differences are caused by their distinctive native behaviors of the root and the children. We will elaborate them when explaining the code. Anyway, I have to claim that the root and the children are implemented in different patterns. Table 7.25 exhibits a new pattern, CC (Cluster Child), for the child.

Pattern: CC (Cluster Child)		
Class	ClusterChild	
Singleton	Yes	
Method	void start()	
	void stop()	
	void setRootIP(IPAddress rootAddress)	
	String getRootIP()	
	int getRootPort()	
	void respondToRoot(ServerMessage response)	
	void broadcastRootIP(RootIPAddressBroadcastNotification notification, int subBranchCount)	
	void broadcastNotify(RootIPAddressBroadcastNotification notification, int subBranchCount)	
	void broadcastRead(HelloWorldBroadcastRequest request, int subBranchCount)	
	void broadcastShutdownNotify(ShutdownChildrenBroadcastNotification notification, int subBranchCount)	
Enclosed Pattern	SD	Peer<ChildDispatcher>
	CBN (Child Broadcasting Notifier)	ClusterChildBroadcastNotifier<RootIPAddressBroadcastNotification, RootIPAddressBroadcastNotificationCreator>
		ClusterChildBroadcastNotifier<HelloWorldBroadcastNotification, HelloWorldBroadcastNotificationCreator>
	CBR (Child Broadcasting Reader)	ClusterChildBroadcastNotifier<ShutdownChildrenBroadcastNotification, ShutdownChildrenBroadcastNotificationCreator>
		ClusterChildBroadcastReader<HelloWorldBroadcastRequest, HelloWorldBroadcastRequestCreator>
Employed API	Peer<Dispatcher extends ServerDispatcher>	
	ClusterChildBroadcastNotifier<Notification extends MulticastMessage, NotificationCreator extends ChildBroadcastNotificationCreatable<Notification>>	
	ClusterChildBroadcastReader<Request extends ClusterBroadcastRequest, RequestCreator extends ChildBroadcastRequestCreatable<Request>>	

Table 7.25 The pattern of CC for the cluster child

Line 35 declares one instance, Peer<ChildDispatcher>, of Peer, which is mainly used to dispatch received data and interact with other nodes actively.

Line 37 declares one instance, rootAddress, of IPAddress, which is used to keep the IP address of the cluster root. In our case, the address is used to send responses to the cluster root after a response corresponding to a multicasting request is generated.

Line 41~42 declares one instance, rootIPBroadcastNotifier, which is used to forward the broadcasting notification of RootIPAddressBroadcastNotification enclosing the IP address of the cluster root, to its children.

Line 45~46 declares one instance, helloWorldBroadcastNotifier, which is used to forward the broadcasting notification of HelloWorldBroadcastNotification to its children.

Line 49~50 declares one instance, helloWorldBroadcastReader, which is used to forward the broadcasting request of HelloWorldBroadcastRequest to its children.

Line 53~54 declares one instance, shutdownBroadcastNotifier, which is used to forward the broadcasting notification of ShutdownChildrenBroadcastNotification to its children.

Line 56~73 defines the cluster child as a singleton.

Line 78~93 defines the method, stop(), which is used to collect all of the resources of the cluster child.

Line 98~167 defines the method, start(). The cluster child is started up if the method is invoked.

Line 101~130 initializes the peer, childPeer, enclosed in the cluster child.

Line 133 starts up the peer, childPeer. Again, a peer has twenty-four parameters in its constructor. If the four ones of ChildDispatcher (Line 110~113) are added, the total number of parameter is twenty-eight. You have to refer to Table 6.21 and Table 5.5.

Line 136~140 initializes the instance of ClusterChildBroadcastNotifier, rootIPBroadcastNotifier. If you are not familiar with the parameters of the constructor, you can refer to Table 7.3.

Line 143~147 initializes the instance of ClusterChildBroadcastNotifier, helloWorldBroadcastNotifier.

Line 150~153 initializes the instance of ClusterChildBroadcastReader, helloWorldBroadcastReader.

Line 157~161 initializes the instance of ClusterChildBroadcastNotifier, shutdownBroadcastNotifier.

In Line 164~166, the peer, childPeer, sends a request of ChatRegistryRequest to the chatting registry server, which we discuss in Section 2 of Chapter 6. If you still remember what we have presented, in the current situation, the child peer accomplishes an application level registry in addition to the system level one on the chatting registry server. If you return to List 7.2 to check, the root peer does not do that. It indicates implicitly that the cluster root is the master that dominates the cluster. In detail, when constructing the hierarchical topology for multicasting, the cluster root is not existed in it. Otherwise, any messages must be sent to itself. That is ridiculous.

Line 172~175 defines the method, setRootIP(), which is invoked when the IP address of the cluster root is received.

Line 180~183 defines the method, getRootIP(), to expose the IP address of the cluster root when it is needed.

Line 188~191 defines the method, getRootPort(), to expose the port of the IP address of the cluster root when needed.

Line 196~199 defines the method, getChildIP(), to expose the IP address of the cluster child.

Line 204~207 defines the method, getChildPort(), to expose the port of the cluster child.

Line 212~215 defines the method, respondToRoot(). It is invoked when a response is created with respect to a new arrived request. If the method is invoked, the response is sent back to the root such that all of responses can be collected to form the final response.

Line 220~224 defines the method, broadcastRootIP(). The cluster child invokes the method when it receives the IP address of the cluster root such that the address can be forwarded to its children to complete the notification broadcasting.

Line 229~233 defines the method, broadcastNotify(). The cluster child invokes the method when it receives the notification of HelloWorldBroadcastNotification such that the notification can be forwarded to its children to complete the notification broadcasting.

Line 238~242 defines the method, broadcastRead(). The cluster child invokes the method when it receives the request of HelloWorldBroadcastRequest such that the request can be forwarded to its children to complete the request broadcasting.

Line 247~252 defines the method, broadcastShutdownNotify(). The cluster child invokes the method when it receives the notification of ShutdownBroadcastNotification such that the notification can be forwarded to its children to complete the notification broadcasting. Different from other broadcasting messages, this is the last message the cluster child needs to forward and process since the cluster child shuts itself down after forwarding the notification.

Well, it has discussed the major code at the client side of the cluster child. You must wonder why there are no anycasting and unicasting APIs in the class. It is easy to explain. The concept of anycasting tells us that the multicasting procedure is terminated if one node at least in the multicasting topology receives the message. So if the current child receives one anycasting message, the termination condition is fulfilled such that the child does not need to forward the message to any others. The concept of unicasting represents another case. The multicasting procedure is terminated if the specified node accepts the message. If the current cluster child receives the message, that means it is the specified node. So the termination condition is fulfilled as well. In either anything or unicasting, it is not necessary for the cluster child to forward the received messages. Hence, you cannot find any multicasting APIs related to anycasting and unicasting in the cluster child. Compared with its counterpart, the cluster root (List 7.2), the difference is really obvious.

3.3 CBN: Child Broadcasting Notifier on CC

CBN (Child Broadcasting Notifier) is another new pattern we encounter when programming a cluster child. It is employed by the most important pattern, CC, at the

cluster child. Its structure is similar to that of MN (Section 2.3). Since the child is required to handle messages of broadcasting only, the pattern is named as broadcasting instead of multicasting.

The CBN contains two components only, i.e., the notification and the notification creator. The notification is the message to be forwarded and the notification creator generates the instance of the notification.

3.3.1 CBN for RootIPAddressBroadcastNotification: rootIPBroadcastNotifier

The instance of ClusterChildBroadcastNotifier is programmed in the pattern CBN, which is exhibited in Table 7.26.

Pattern: CBN (Child Broadcasting Notifier)		
Class	ClusterChildBroadcastNotifier<RootIPAddressBroadcastNotification, RootIPAddressBroadcastNotificationCreator>	
Singleton	No	
Method	void notify(RootIPAddressBroadcastNotification notification, int subBranchCount)	
	void dispose()	
Enclosed Pattern	BN (Broadcast Notification)	RootIPAddressBroadcastNotification
	BNC (Broadcast Notification Creator)	RootIPAddressBroadcastNotificationCreator
Employed API	Serializable	
	MulticastMessage	
	ClusterChildBroadcastNotifier<Notification extends MulticastMessage, NotificationCreator extends ChildBroadcastNotificationCreatable<Notification>>	
	RootBroadcastNotificationCreatable<Notification extends MulticastMessage, NotificationData extends Serializable>	

Table 7.26 The pattern of CBN for the notification of RootIPAddressBroadcastNotification

The notification for CBN is RootIPAddressBroadcastNotification, which is the same as the one in List 7.4. The notification is always forwarded within the cluster until n children IP addresses are available in the message.

The notification creator, RootIPAddressBroadcastNotificationCreator, of CBN is shown in List 7.36.

```
1    package com.greatfree.cluster.message.child;
2
3    import java.util.HashMap;
4
5    import com.greatfree.cluster.message.RootIPAddressBroadcastNotification;
6    import com.greatfree.message.ChildBroadcastNotificationCreatable;
7    import com.greatfree.util.IPAddress;
8    import com.greatfree.util.Tools;
9
10   /*
11    * The creator generates the notifications of RootIPAddressBroadcastNotification that should be broadcast to
12    * the current distributed node's children. 05/12/2017, Bing Li
13    */
14
15   // Created: 05/20/2017, Bing Li
16   public class RootIPAddressBroadcastNotificationCreator implements ChildBroadcastNotificationCreatable
17       <RootIPAddressBroadcastNotification>
18   {
19       @Override
20       public RootIPAddressBroadcastNotification createInstanceWithChildren
21           (RootIPAddressBroadcastNotification msg, HashMap<String, IPAddress> children)
22       {
23           return new RootIPAddressBroadcastNotification(Tools.generateUniqueKey(), children,
24               msg.getRootAddress());
25       }
```

```
26    }
```

List 7.36 The code of RootIPAddressBroadcastNotificationCreator.java

3.3.2 CBN for HelloWorldBroadcastNotification: helloWorldBroadcastNotifier

The CBN, helloWorldBroadcastNotifier, forwards the notification of HelloWorldBroadcastNotification to its children.

The notification is identical to the one in List 7.7.

The notification creator is shown in List 7.37.

```
1     package com.greatfree.cluster.message.child;
2
3     import java.util.HashMap;
4
5     import com.greatfree.cluster.message.HelloWorldBroadcastNotification;
6     import com.greatfree.message.ChildBroadcastNotificationCreatable;
7     import com.greatfree.util.IPAddress;
8     import com.greatfree.util.Tools;
9
10    /*
11     * The creator generates the notifications of HelloWorldNotification that should be broadcast to the current
12     * distributed node's children. 05/12/2017, Bing Li
13     */
14
15    // Created: 05/12/2017, Bing Li
16    public class HelloWorldBroadcastNotificationCreator implements ChildBroadcastNotificationCreatable
17        <HelloWorldBroadcastNotification>
18    {
19        @Override
20        public HelloWorldBroadcastNotification createInstanceWithChildren(HelloWorldBroadcastNotification
21            msg, HashMap<String, IPAddress> children)
22        {
23            return new HelloWorldBroadcastNotification(Tools.generateUniqueKey(), children,
24                msg.getHelloWorld());
25        }
26    }
```

List 7.37 The code of HelloWorldBroadcastNotificationCreator.java

3.3.3 CBN for ShutdownChildrenBroadcastNotification: shutdownBroadcastNotifier

The notification of the CBN is ShutdownChildrenBroadcastNotification, as shown in List 7.13.

The notification creator, ShutdownChildrenBroadcastNotificationCreator, is presented in List 7.38.

```
1     package com.greatfree.cluster.message.child;
2
3     import java.util.HashMap;
4
5     import com.greatfree.cluster.message.ShutdownChildrenBroadcastNotification;
6     import com.greatfree.message.ChildBroadcastNotificationCreatable;
7     import com.greatfree.util.IPAddress;
8     import com.greatfree.util.Tools;
9
10    /*
11     * The creator generates the notifications of ShutdownChildrenBroadcastNotification that should be broadcast
12     * to the current distributed node's children. 05/12/2017, Bing Li
```

```
13     */
14
15     // Created: 05/19/2017, Bing Li
16     public class ShutdownChildrenBroadcastNotificationCreator implements
17         ChildBroadcastNotificationCreatable<ShutdownChildrenBroadcastNotification>
18     {
19         @Override
20         public ShutdownChildrenBroadcastNotification createInstanceWithChildren
21             (ShutdownChildrenBroadcastNotification msg, HashMap<String, IPAddress> children)
22         {
23             return new ShutdownChildrenBroadcastNotification(Tools.generateUniqueKey(), children);
24         }
25     }
```

List 7.38 The code of ShutdownChildrenBroadcastNotificationCreator.java

3.4 CBR: Child Broadcasting Reader on CC

For the particular nature of the cluster child, the only task for it is to forward requests to its children as soon as possible. No responses are replied to it even though requests have to be forwarded by it. From this point of view, the Child Broadcasting Reader (CBR) at the client side of the cluster child is identical to the CBN inherently. However, we are doing something in the area of Software Engineering such that one of the most important rules is to comfort developers. For that, we offer the pattern another name, i.e., Child Broadcasting Reader (CBR), since the pattern needs to forward requests rather than notifications.

CBR has two components, i.e., the request and the request creator. Compared with its counterpart, MR, on the cluster root, CBR is really a different stuff. At least, the pattern does not have a component of the response. Thus, it does nothing on that, for example, receiving the response. This reminds us again that the pattern is actually a notifier rather than a reader.

In the code of ClusterChild in List 7.35, only one CBR is employed, i.e., the one for HelloWorldBroadcastRequest declared at Line 49~50 of List 7.35.

The request is defined in List 7.15.

The request creator, HelloWorldBroadcastRequestCreator, for the CBR on the cluster child is exhibited in List 7.39.

```
1      package com.greatfree.cluster.message.child;
2
3      import java.util.HashMap;
4
5      import com.greatfree.cluster.message.HelloWorldBroadcastRequest;
6      import com.greatfree.message.ChildBroadcastRequestCreatable;
7      import com.greatfree.util.IPAddress;
8      import com.greatfree.util.Tools;
9
10     /*
11      * The creator generates the notifications of HelloWorldBroadcastRequest that should be broadcast to the
12      * current distributed node's children. 05/20/2017, Bing Li
13      */
14
15     // Created: 05/21/2017, Bing Li
16     public class HelloWorldBroadcastRequestCreator implements ChildBroadcastRequestCreatable
17         <HelloWorldBroadcastRequest>
18     {
19         @Override
20         public HelloWorldBroadcastRequest createInstanceWithChildren(HelloWorldBroadcastRequest msg,
```

```
21              HashMap<String, IPAddress> children)
22          {
23              return new HelloWorldBroadcastRequest(msg.getHelloWorld(), Tools.generateUniqueKey(),
24                  msg.getCollaboratorKey(), children);
25          }
26      }
```

List 7.39 The code of HelloWorldBroadcastRequestCreator.java

3.5 SD: Programming the Server Side of the Cluster Child

With the server side, a distributed node over the Internet cannot receive any messages. As a node within a cluster, it should receive data definitely. So we need to program the server side for the cluster child. When you program the CC for the node, it declares one instance, childPeer, of Peer at Line 35 of ClusterChild in List 7.35. The childPeer contains the code to process incoming message. You must be familiar with it a lot since it follows exactly the pattern of SD as shown in Table 7.27.

Pattern: SD (Server Dispatcher)		
Class	Peer<ChildDispatcher>	
Singleton	No	
Method	void consume(OutMessageStream<ServerMessage> message)	
	void shutdown()	
Enclosed Pattern	BND (Bound Notification Dispatcher)	BoundNotificationDispatcher<RootIPAddressBroadcastNotification, ClusterMessageDisposer<RootIPAddressBroadcastNotification>, RootIPAddressBroadcastNotificationThread, RootIPAddressBroadcastNotificationThreadCreator>
		BoundNotificationDispatcher<RootIPAddressBroadcastNotification, ClusterMessageDisposer<RootIPAddressBroadcastNotification>, BroadcastRootIPAddressNotificationThread, BroadcastRootIPAddressNotificationThreadCreator>
		BoundNotificationDispatcher<HelloWorldBroadcastNotification, ClusterMessageDisposer<HelloWorldBroadcastNotification>, HelloWorldBroadcastNotificationThread, HelloWorldBroadcastNotificationThreadCreator>
		BoundNotificationDispatcher<HelloWorldBroadcastNotification, ClusterMessageDisposer<HelloWorldBroadcastNotification>, BroadcastHelloWorldNotificationThread, BroadcastHelloWorldNotificationThreadCreator>
		BoundNotificationDispatcher<HelloWorldBroadcastRequest, ClusterMessageDisposer<HelloWorldBroadcastRequest>, BroadcastHelloWorldRequestThread, BroadcastHelloWorldRequestThreadCreator>
	BRD (Bound Request Dispatcher)	BoundRequestDispatcher<HelloWorldBroadcastRequest, HelloWorldBroadcastResponse, ClusterMessageDisposer<HelloWorldBroadcastRequest>, HelloWorldBroadcastRequestThread, HelloWorldBroadcastRequestThreadCreator>
	ND (Notification Dispatcher)	NotificationDispatcher<HelloWorldAnycastNotification, HelloWorldAnycastNotificationThread, HelloWorldAnycastNotificationThreadCreator>
		NotificationDispatcher<HelloWorldUnicastNotification, HelloWorldUnicastNotificationThread, HelloWorldUnicastNotificationThreadCreator>
		NotificationDispatcher<HelloWorldUnicastRequest, HelloWorldUnicastRequestThread, HelloWorldUnicastRequestThreadCreator>
		NotificationDispatcher<ShutdownChildrenBroadcastNotification, ShutdownChildrenBroadcastNotificationThread, ShutdownChildrenBroadcastNotificationThreadCreator>
Employed API	ServerDispatcher	
	NotificationDispatcher<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>, ThreadCreator extends NotificationThreadCreatable<Notification, NotificationThread>>	
	ClusterMessageDisposer<Message extends MulticastMessage>	
	BoundNotificationDispatcher<Notification extends MulticastMessage, Binder extends MessageBindable<Notification>, NotificationThread extends BoundNotificationQueue<Notification, Binder>, ThreadCreator extends BoundNotificationThreadCreatable<Notification, Binder, NotificationThread>>	
	BoundRequestDispatcher<Request extends MulticastMessage, Response extends ServerMessage, RequestBinder extends MessageBindable<Request>, RequestThread extends BoundRequestQueue<Request, Response, RequestBinder>, RequestThreadCreator extends BoundRequestThreadCreatable<Request, Response, RequestBinder, RequestThread>>	

Table 7.27 The pattern SD for the cluster child

I hope you still love GreatFree at this moment. According to Table 7.27, you get two new patterns, the Bound Notification Dispatcher (BND) and the Bound Request Dispatcher (BRD). Before explaining them, let us present the code, ChildDispatcher.java, first as shown in List 7.40.

```
1   package com.greatfree.cluster.child;
2
3   import java.util.Calendar;
4
5   import com.greatfree.cluster.message.ClusterMessageDisposer;
6   import com.greatfree.cluster.message.ClusterMessageType;
7   import com.greatfree.cluster.message.HelloWorldAnycastNotification;
8   import com.greatfree.cluster.message.HelloWorldAnycastRequest;
9   import com.greatfree.cluster.message.HelloWorldBroadcastNotification;
10  import com.greatfree.cluster.message.HelloWorldBroadcastRequest;
11  import com.greatfree.cluster.message.HelloWorldBroadcastResponse;
12  import com.greatfree.cluster.message.HelloWorldUnicastNotification;
13  import com.greatfree.cluster.message.HelloWorldUnicastRequest;
14  import com.greatfree.cluster.message.RootIPAddressBroadcastNotification;
15  import com.greatfree.cluster.message.ShutdownChildrenBroadcastNotification;
16  import com.greatfree.concurrency.BoundRequestDispatcher;
17  import com.greatfree.concurrency.BoundNotificationDispatcher;
18  import com.greatfree.concurrency.NotificationDispatcher;
19  import com.greatfree.message.ServerMessage;
20  import com.greatfree.remote.OutMessageStream;
21  import com.greatfree.server.ServerDispatcher;
22  import com.greatfree.testing.data.ServerConfig;
23
24  /*
25   * The dispatcher serves as a message dispatcher for messages within the cluster. 05/10/2017, Bing Li
26   */
27
28  // Created: 05/11/2017, Bing Li
29  public class ChildDispatcher extends ServerDispatcher
30  {
31      // The disposer is the binder that synchronizes the two bound notification dispatchers,
32      // processRootIPBroadcastNotificationDispatcher and broadcastRootIPNotificationDispatcher. After both of
33      // them finish their respective tasks concurrently, it disposes the notification of
34      // RootIPAddressBroadcastNotification finally. 11/27/2014, Bing Li
35      private ClusterMessageDisposer<RootIPAddressBroadcastNotification> rootIPNotificationDisposer;
36      // The dispatcher processes the notification of RootIPAddressBroadcastNotification. It must be
37      // synchronized by the binder, broadcastNotificationDisposer. So it is implemented as a bound notification
38      // dispatcher. 11/27/2014, Bing Li
39      private BoundNotificationDispatcher<RootIPAddressBroadcastNotification, ClusterMessageDisposer
40          <RootIPAddressBroadcastNotification>, RootIPAddressBroadcastNotificationThread,
41          RootIPAddressBroadcastNotificationThreadCreator> processRootIPBroadcastNotificationDispatcher;
42      // The dispatcher to disseminate the notification of RootIPAddressBroadcastNotification to children nodes. It
43      // must be synchronized by the binder, broadcastNotificationDisposer. So it is implemented as a bound
44      // notification dispatcher. 11/27/2014, Bing Li
45      private BoundNotificationDispatcher<RootIPAddressBroadcastNotification, ClusterMessageDisposer
46          <RootIPAddressBroadcastNotification>, BroadcastRootIPAddressNotificationThread,
47          BroadcastRootIPAddressNotificationThreadCreator> broadcastRootIPNotificationDispatcher;
48
49      // The disposer is the binder that synchronizes the two bound notification dispatchers,
50      // processHelloBroadcastNotificationDispatcher and broadcastHelloWorldNotificationDispatcher. After both
51      // of them finish their respective tasks concurrently, it disposes the notification of HelloWorldNotification
52      // finally. 11/27/2014, Bing Li
53      private ClusterMessageDisposer<HelloWorldBroadcastNotification> helloNotificationDisposer;
54      // The dispatcher processes the notification of HelloWorldBroadcastNotification. It must be
55      // synchronized by the binder, broadcastNotificationDisposer. So it is implemented as a bound notification
56      // dispatcher. 11/27/2014, Bing Li
57      private BoundNotificationDispatcher<HelloWorldBroadcastNotification, ClusterMessageDisposer
58          <HelloWorldBroadcastNotification>, HelloWorldBroadcastNotificationThread,
59          HelloWorldBroadcastNotificationThreadCreator> processHelloBroadcastNotificationDispatcher;
60      // The dispatcher to disseminate the notification of HelloWorldBroadcastNotification to children nodes. It
61      // must be synchronized by the binder, broadcastNotificationDisposer. So it is implemented as a bound
62      // notification dispatcher. 11/27/2014, Bing Li
63      private BoundNotificationDispatcher<HelloWorldBroadcastNotification, ClusterMessageDisposer
64          <HelloWorldBroadcastNotification>, BroadcastHelloWorldNotificationThread,
65          BroadcastHelloWorldNotificationThreadCreator> broadcastHelloWorldNotificationDispatcher;
66
67      // Declare a notification dispatcher to process anycast messages to the child when such a notification is
68      // received. 05/19/2016, Bing Li
69      private NotificationDispatcher<HelloWorldAnycastNotification, HelloWorldAnycastNotificationThread,
```

```
70          HelloWorldAnycastNotificationThreadCreator> processHelloWorldAnycastNotificationDispatcher;
71
72      // Declare a notification dispatcher to process unicast messages to the child when such a notification is
73      // received. 05/19/2016, Bing Li
74      private NotificationDispatcher<HelloWorldUnicastNotification, HelloWorldUnicastNotificationThread,
75          HelloWorldUnicastNotificationThreadCreator> processHelloWorldUnicastNotificationDispatcher;
76
77      // Declare the message disposer to collect requests after it is handled and broadcast. 01/14/2016, Bing Li
78      private ClusterMessageDisposer<HelloWorldBroadcastRequest> helloRequestDisposer;
79      // Declare the dispatcher to handle the broadcast request. 01/14/2016, Bing Li
80      private BoundRequestDispatcher<HelloWorldBroadcastRequest, HelloWorldBroadcastResponse,
81          ClusterMessageDisposer<HelloWorldBroadcastRequest>, HelloWorldBroadcastRequestThread,
82          HelloWorldBroadcastRequestThreadCreator> processHelloWorldBroadcastRequestDispatcher;
83      // Declare the broadcast dispatcher to broadcast the request. 01/14/2016, Bing Li
84      private BoundNotificationDispatcher<HelloWorldBroadcastRequest, ClusterMessageDisposer
85          <HelloWorldBroadcastRequest>, BroadcastHelloWorldRequestThread,
86          BroadcastHelloWorldRequestThreadCreator> multicastBroadcastRequestDispatcher;
87
88      // Declare a notification dispatcher to process anycast requests to the child when such a
89      // notification is received. 05/19/2016, Bing Li
90      private NotificationDispatcher<HelloWorldAnycastRequest, HelloWorldAnycastRequestThread,
91          HelloWorldAnycastRequestThreadCreator> processHelloWorldAnycastRequestDispatcher;
92
93      // Declare a notification dispatcher to process unicast requests to the child when such a
94      // notification is received. 05/19/2016, Bing Li
95      private NotificationDispatcher<HelloWorldUnicastRequest, HelloWorldUnicastRequestThread,
96          HelloWorldUnicastRequestThreadCreator> processHelloWorldUnicastRequestDispatcher;
97
98      // The dispatcher processes the notification of ShutdownChildrenBroadcastNotification. It must be
99      // synchronized by the binder, broadcastNotificationDisposer. So it is implemented as a bound notification
100     // dispatcher. 11/27/2014, Bing Li
101     private NotificationDispatcher<ShutdownChildrenBroadcastNotification,
102         ShutdownChildrenBroadcastNotificationThread,
103         ShutdownChildrenBroadcastNotificationThreadCreator>
104         processShutdownBroadcastNotificationDispatcher;
105
106     /*
107      * Initialize the dispatcher. 11/25/2014, Bing Li
108      */
109     public ChildDispatcher(int threadPoolSize, long threadKeepAliveTime, int schedulerPoolSize, long
110         schedulerKeepAliveTime)
111     {
112         super(threadPoolSize, threadKeepAliveTime, schedulerPoolSize, schedulerKeepAliveTime);
114
115         // Initialize the disposer for the notification of RootIPAddressBroadcastNotification, which
116         // works as a binder. 11/27/2014, Bing Li
117         this.rootIPNotificationDisposer = new ClusterMessageDisposer
118             <RootIPAddressBroadcastNotification>();
119
120         // Initialize the bound notification dispatcher for the notification,
121         // RootIPAddressBroadcastNotification. 11/27/2014, Bing Li
122         this.processRootIPBroadcastNotificationDispatcher = new BoundNotificationDispatcher
123             .BoundNotificationDispatcherBuilder<RootIPAddressBroadcastNotification,
124             ClusterMessageDisposer<RootIPAddressBroadcastNotification>,
125             RootIPAddressBroadcastNotificationThread, RootIPAddressBroadcastNotificationThreadCreator>()
126             .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
127             .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
128             .binder(this.rootIPNotificationDisposer)
129             .threadCreator(new RootIPAddressBroadcastNotificationThreadCreator())
130             .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
131             .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
132             .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
133             .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
134             .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
135             .scheduler(super.getSchedulerPool())
136             .build();
137
138         // Initialize the bound notification dispatcher for the notification, RootIPAddressBroadcastNotification, to
139         // disseminate the notification to children crawlers. 11/27/2014, Bing Li
140         this.broadcastRootIPNotificationDispatcher = new BoundNotificationDispatcher
141             .BoundNotificationDispatcherBuilder<RootIPAddressBroadcastNotification,
142             ClusterMessageDisposer<RootIPAddressBroadcastNotification>,
143             BroadcastRootIPAddressNotificationThread, BroadcastRootIPAddressNotificationThreadCreator>()
144             .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
145             .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
146             .binder(this.rootIPNotificationDisposer)
147             .threadCreator(new BroadcastRootIPAddressNotificationThreadCreator())
```

```
148             .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
149             .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
150             .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
151             .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
152             .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
153             .scheduler(super.getSchedulerPool())
154             .build();
155
156         // Initialize the disposer for the notification of HelloWorldNotification, which works as
157         // a binder. 11/27/2014, Bing Li
158         this.helloNotificationDisposer = new ClusterMessageDisposer<HelloWorldBroadcastNotification>();
159
160         // Initialize the bound notification dispatcher for the notification,
161         // HelloWorldBroadcastNotification. 11/27/2014, Bing Li
162         this.processHelloBroadcastNotificationDispatcher = new BoundNotificationDispatcher
163             .BoundNotificationDispatcherBuilder<HelloWorldBroadcastNotification, ClusterMessageDisposer
164             <HelloWorldBroadcastNotification>, HelloWorldBroadcastNotificationThread,
165             HelloWorldBroadcastNotificationThreadCreator>()
166             .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
167             .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
168             .binder(this.helloNotificationDisposer)
169             .threadCreator(new HelloWorldBroadcastNotificationThreadCreator())
170             .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
171             .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
172             .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
173             .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
174             .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
175             .scheduler(super.getSchedulerPool())
176             .build();
177
178         // Initialize the bound notification dispatcher for the notification, BroadcastNotification, to
179         // disseminate the notification to children crawlers. 11/27/2014, Bing Li
180         this.broadcastHelloWorldNotificationDispatcher = new BoundNotificationDispatcher
181             .BoundNotificationDispatcherBuilder<HelloWorldBroadcastNotification, ClusterMessageDisposer
182             <HelloWorldBroadcastNotification>, BroadcastHelloWorldNotificationThread,
183             BroadcastHelloWorldNotificationThreadCreator>()
184             .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
185             .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
186             .binder(this.helloNotificationDisposer)
187             .threadCreator(new BroadcastHelloWorldNotificationThreadCreator())
188             .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
189             .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
190             .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
191             .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
192             .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
193             .scheduler(super.getSchedulerPool())
194             .build();
195
196         // Initialize the notification dispatcher for the notification,
197         // HelloWorldAnycastNotification. 11/27/2014, Bing Li
198         this.processHelloWorldAnycastNotificationDispatcher = new NotificationDispatcher
199             .NotificationDispatcherBuilder<HelloWorldAnycastNotification,
200             HelloWorldAnycastNotificationThread, HelloWorldAnycastNotificationThreadCreator>()
201             .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
202             .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
203             .threadCreator(new HelloWorldAnycastNotificationThreadCreator())
204             .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
205             .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
206             .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
207             .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
208             .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
209             .scheduler(super.getSchedulerPool())
210             .build();
211
212         // Initialize the notification dispatcher for the notification,
213         // HelloWorldAnycastNotification. 11/27/2014, Bing Li
214         this.processHelloWorldUnicastNotificationDispatcher = new NotificationDispatcher
215             .NotificationDispatcherBuilder<HelloWorldUnicastNotification, HelloWorldUnicastNotificationThread,
216             HelloWorldUnicastNotificationThreadCreator>()
217             .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
218             .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
219             .threadCreator(new HelloWorldUnicastNotificationThreadCreator())
220             .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
221             .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
222             .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
223             .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
224             .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
```

```
225             .scheduler(super.getSchedulerPool())
226             .build();
227
228         // Initialize the disposer for the notification of HelloWorldBroadcastRequest,
229         // which works as a binder. 11/27/2014, Bing Li
230         this.helloRequestDisposer = new ClusterMessageDisposer<HelloWorldBroadcastRequest>();
231
232         // Initialize the dispatcher to handle the request. 01/14/2016, Bing Li
233         this.processHelloWorldBroadcastRequestDispatcher = new BoundRequestDispatcher
234             .BoundRequestDispatcherBuilder<HelloWorldBroadcastRequest, HelloWorldBroadcastResponse,
235             ClusterMessageDisposer<HelloWorldBroadcastRequest>, HelloWorldBroadcastRequestThread,
236             HelloWorldBroadcastRequestThreadCreator>()
237             .threadPoolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
238             .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
239             .binder(this.helloRequestDisposer)
240             .threadCreator(new HelloWorldBroadcastRequestThreadCreator())
241             .maxTaskSize(ServerConfig.MAX_REQUEST_TASK_SIZE)
242             .dispatcherWaitTime(ServerConfig.REQUEST_DISPATCHER_WAIT_TIME)
243             .waitRound(ServerConfig.REQUEST_DISPATCHER_WAIT_ROUND)
244             .idleCheckDelay(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_DELAY)
245             .idleCheckPeriod(ServerConfig.REQUEST_DISPATCHER_IDLE_CHECK_PERIOD)
246             .scheduler(super.getSchedulerPool())
247             .build();
248
249         // Initialize the dispatcher to broadcast the request. 05/20/2017, Bing Li
250         this.multicastBroadcastRequestDispatcher = new BoundNotificationDispatcher
251             .BoundNotificationDispatcherBuilder<HelloWorldBroadcastRequest, ClusterMessageDisposer
252             <HelloWorldBroadcastRequest>, BroadcastHelloWorldRequestThread,
253             BroadcastHelloWorldRequestThreadCreator>()
254             .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
255             .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
256             .binder(this.helloRequestDisposer)
257             .threadCreator(new BroadcastHelloWorldRequestThreadCreator())
258             .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
259             .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
260             .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
261             .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
262             .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
263             .scheduler(super.getSchedulerPool())
264             .build();
265
266         // Initialize the notification dispatcher for the notification,
267         // HelloWorldAnycastRequest. 11/27/2014, Bing Li
268         this.processHelloWorldAnycastRequestDispatcher = new NotificationDispatcher
269             .NotificationDispatcherBuilder<HelloWorldAnycastRequest, HelloWorldAnycastRequestThread,
270             HelloWorldAnycastRequestThreadCreator>()
271             .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
272             .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
273             .threadCreator(new HelloWorldAnycastRequestThreadCreator())
274             .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
275             .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
276             .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
277             .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
278             .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
279             .scheduler(super.getSchedulerPool())
280             .build();
281
282         // Initialize the notification dispatcher for the notification, HelloWorldUnicastRequest. 11/27/2014, Bing Li
283         this.processHelloWorldUnicastRequestDispatcher = new NotificationDispatcher
284             .NotificationDispatcherBuilder<HelloWorldUnicastRequest, HelloWorldUnicastRequestThread,
285             HelloWorldUnicastRequestThreadCreator>()
286             .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
287             .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
288             .threadCreator(new HelloWorldUnicastRequestThreadCreator())
289             .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
290             .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
291             .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
292             .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
293             .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
294             .scheduler(super.getSchedulerPool())
295             .build();
296
297         // Initialize the notification dispatcher for the notification,
298         // HelloWorldAnycastNotification. 11/27/2014, Bing Li
299         this.processShutdownBroadcastNotificationDispatcher = new NotificationDispatcher
300             .NotificationDispatcherBuilder<ShutdownChildrenBroadcastNotification,
301             ShutdownChildrenBroadcastNotificationThread,
```

```
302             ShutdownChildrenBroadcastNotificationThreadCreator>()
303             .poolSize(ServerConfig.NOTIFICATION_DISPATCHER_POOL_SIZE)
304             .keepAliveTime(ServerConfig.NOTIFICATION_DISPATCHER_THREAD_ALIVE_TIME)
305             .threadCreator(new ShutdownChildrenBroadcastNotificationThreadCreator())
306             .maxTaskSize(ServerConfig.MAX_NOTIFICATION_TASK_SIZE)
307             .dispatcherWaitTime(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_TIME)
308             .waitRound(ServerConfig.NOTIFICATION_DISPATCHER_WAIT_ROUND)
309             .idleCheckDelay(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_DELAY)
310             .idleCheckPeriod(ServerConfig.NOTIFICATION_DISPATCHER_IDLE_CHECK_PERIOD)
311             .scheduler(super.getSchedulerPool())
312             .build();
313     }
314
315     /*
316     * Shutdown the dispatcher. 11/25/2014, Bing Li
317      */
318     public void shutdown() throws InterruptedException
319     {
320         // Shutdown the bound notification dispatcher to process broadcast notifications. 11/27/2014, Bing Li
321         this.processRootIPBroadcastNotificationDispatcher.dispose();
322         // Shutdown the bound notification dispatcher for disseminating the broadcast
323         // notifications. 11/27/2014, Bing Li
324         this.broadcastRootIPNotificationDispatcher.dispose();
325         // Shutdown the bound notification dispatcher to process broadcast notifications. 11/27/2014, Bing Li
326         this.processHelloBroadcastNotificationDispatcher.dispose();
327         // Shutdown the bound notification dispatcher for disseminating the broadcast
328         // notifications. 11/27/2014, Bing Li
329         this.broadcastHelloWorldNotificationDispatcher.dispose();
330         // Shutdown the notification dispatcher to process anycast notifications. 11/27/2014, Bing Li
331         this.processHelloWorldAnycastNotificationDispatcher.dispose();
332         // Shutdown the notification dispatcher to process unicast notifications. 11/27/2014, Bing Li
333         this.processHelloWorldUnicastNotificationDispatcher.dispose();
334         // Shutdown the request dispatcher that handles the request. 05/21/2017, Bing Li
335         this.processHelloWorldBroadcastRequestDispatcher.dispose();
336         // Shutdown the request dispatcher that multicast the request. 05/21/2017, Bing Li
337         this.multicastBroadcastRequestDispatcher.dispose();
338         // Shutdown the bound notification dispatcher to process anycast requests. 11/27/2014, Bing Li
339         this.processHelloWorldAnycastRequestDispatcher.dispose();
340         // Shutdown the bound notification dispatcher to process unicast requests. 11/27/2014, Bing Li
341         this.processHelloWorldUnicastRequestDispatcher.dispose();
342         // Shutdown the bound notification dispatcher to process broadcast notifications. 11/27/2014, Bing Li
343         this.processShutdownBroadcastNotificationDispatcher.dispose();
344         // Shutdown the dispatcher. 05/20/2017, Bing Li
345         super.shutdown();
346     }
347
348     /*
349     * Dispatch received messages to corresponding threads respectively for
350     * concurrent processing. 11/25/2014, Bing Li
351     */
352     public void consume(OutMessageStream<ServerMessage> message)
353     {
354         HelloWorldBroadcastNotification helloWorldNotification;
355         RootIPAddressBroadcastNotification rootIPNotification;
356         HelloWorldBroadcastRequest helloWorldRequest;
357         switch (message.getMessage().getType())
358         {
359             case ClusterMessageType.ROOT_IPADDRESS_BROADCAST_NOTIFICATION:
360                 System.out.println("<" + ClusterChild.CLUSTER().getChildIP() + ":" + ClusterChild.CLUSTER()
361                     .getChildPort() + "> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @" +
362                     Calendar.getInstance().getTime());
363                 // Cast the message. 11/27/2014, Bing Li
364                 rootIPNotification = (RootIPAddressBroadcastNotification)message.getMessage();
365
366                 // Check whether the bound broadcast notification dispatcher is ready. 02/02/2016, Bing Li
367                 if (!this.processRootIPBroadcastNotificationDispatcher.isReady())
368                 {
369                     // Execute the bound broadcast notification dispatcher. 02/02/2016, Bing Li
370                     super.execute(this.processRootIPBroadcastNotificationDispatcher);
371                 }
372                 // Enqueue the notification into those bound broadcast notification dispatchers.
373                 // The notifications are queued and processed asynchronously with a proper
374                 // synchronization. 11/27/2014, Bing Li
375                 this.processRootIPBroadcastNotificationDispatcher.enqueue(rootIPNotification);
376
377                 // Check whether the bound broadcast notification dispatcher is ready. 02/02/2016, Bing Li
378                 if (!this.broadcastRootIPNotificationDispatcher.isReady())
```

```
379                 {
380                     // Execute the bound notification broadcast dispatcher. 02/02/2016, Bing Li
381                     super.execute(this.broadcastRootIPNotificationDispatcher);
382                 }
383                 // Enqueue the notification into those bound broadcast notification dispatchers. The notifications
384                 // are queued and processed asynchronously with a proper synchronization. 11/27/2014, Bing Li
385                 this.broadcastRootIPNotificationDispatcher.enqueue(rootIPNotification);
386                 break;
387
388             // Process the notification of HelloWorldBroadcastNotification. 11/27/2014, Bing Li
389             case ClusterMessageType.HELLO_WORLD_BROADCAST_NOTIFICATION:
390                 System.out.println("<" + ClusterChild.CLUSTER().getChildIP() + ":" + ClusterChild.CLUSTER()
391                     .getChildPort() + "> HELLO_WORLD_BROADCAST_NOTIFICATION received @" +
392                     Calendar.getInstance().getTime());
393                 // Cast the message. 11/27/2014, Bing Li
394                 helloWorldNotification = (HelloWorldBroadcastNotification)message.getMessage();
395
396                 // Check whether the bound broadcast notification dispatcher is ready. 02/02/2016, Bing Li
397                 if (!this.processHelloBroadcastNotificationDispatcher.isReady())
398                 {
399                     // Execute the bound broadcast notification dispatcher. 02/02/2016, Bing Li
400                     super.execute(this.processHelloBroadcastNotificationDispatcher);
401                 }
402                 // Enqueue the notification into those bound broadcast notification dispatchers. The notifications
403                 // are queued and processed asynchronously with a proper synchronization. 11/27/2014, Bing Li
404                 this.processHelloBroadcastNotificationDispatcher.enqueue(helloWorldNotification);
405
406                 // Check whether the bound broadcast notification dispatcher is ready. 02/02/2016, Bing Li
407                 if (!this.broadcastHelloWorldNotificationDispatcher.isReady())
408                 {
409                     // Execute the bound notification broadcast dispatcher. 02/02/2016, Bing Li
410                     super.execute(this.broadcastHelloWorldNotificationDispatcher);
411                 }
412                 // Enqueue the notification into those bound broadcast notification dispatchers. The notifications
413                 // are queued and processed asynchronously with a proper synchronization. 11/27/2014, Bing Li
414                 this.broadcastHelloWorldNotificationDispatcher.enqueue(helloWorldNotification);
415                 break;
416
417             // Process the notification of HelloWorldAnycastNotification. 11/27/2014, Bing Li
418             case ClusterMessageType.HELLO_WORLD_ANYCAST_NOTIFICATION:
419                 System.out.println("<" + ClusterChild.CLUSTER().getChildIP() + ":" + ClusterChild.CLUSTER()
420                     .getChildPort() + "> HELLO_WORLD_ANYCAST_NOTIFICATION received @" +
421                     Calendar.getInstance().getTime());
422                 // Check whether the anycast notification dispatcher is ready. 02/02/2016, Bing Li
423                 if (!this.processHelloWorldAnycastNotificationDispatcher.isReady())
424                 {
425                     // Execute the anycast notification dispatcher. 02/02/2016, Bing Li
426                     super.execute(this.processHelloWorldAnycastNotificationDispatcher);
427                 }
428                 // Enqueue the notification into those anycast notification dispatchers. The notifications are
429                 // queued and processed asynchronously with a proper synchronization. 11/27/2014, Bing Li
430                 this.processHelloWorldAnycastNotificationDispatcher.enqueue((HelloWorldAnycastNotification)
431                     message.getMessage());
432                 break;
433
434             case ClusterMessageType.HELLO_WORLD_UNICAST_NOTIFICATION:
435                 System.out.println("<" + ClusterChild.CLUSTER().getChildIP() + ":" + ClusterChild.CLUSTER()
436                     .getChildPort() + "> HELLO_WORLD_UNICAST_NOTIFICATION received @" +
437                     Calendar.getInstance().getTime());
438                 // Check whether the unicast notification dispatcher is ready. 02/02/2016, Bing Li
439                 if (!this.processHelloWorldUnicastNotificationDispatcher.isReady())
440                 {
441                     // Execute the unicast notification dispatcher. 02/02/2016, Bing Li
442                     super.execute(this.processHelloWorldUnicastNotificationDispatcher);
443                 }
444                 // Enqueue the notification into those unicast notification dispatchers. The notifications are
445                 // queued and processed asynchronously with a proper synchronization. 11/27/2014, Bing Li
446                 this.processHelloWorldUnicastNotificationDispatcher.enqueue((HelloWorldUnicastNotification)
447                     message.getMessage());
448                 break;
449
450             case ClusterMessageType.HELLO_WORLD_BROADCAST_REQUEST:
451                 System.out.println("<" + ClusterChild.CLUSTER().getChildIP() + ":" + ClusterChild.CLUSTER()
452                     .getChildPort() + "> HELLO_WORLD_BROADCAST_REQUEST received @" +
453                     Calendar.getInstance().getTime());
454                 // Get the broadcast request. 05/20/2017, Bing Li
455                 helloWorldRequest = (HelloWorldBroadcastRequest)message.getMessage();
```

```
456                 // Check whether the broadcast request processing dispatcher is ready. 05/21/2017, Bing Li
457                 if (!this.processHelloWorldBroadcastRequestDispatcher.isReady())
458                 {
459                     // Execute the broadcast request processing dispatcher. 05/21/2017, Bing Li
460                     super.execute(this.processHelloWorldBroadcastRequestDispatcher);
461                 }
462                 // Enqueue the broadcast request into the broadcast request processing
463                 // dispatcher. 05/21/2017, Bing Li
464                 this.processHelloWorldBroadcastRequestDispatcher.enqueue(helloWorldRequest);
465
466                 // Check whether the broadcast request multicasting dispatcher is ready. 05/21/2017, Bing Li
467                 if (!this.multicastBroadcastRequestDispatcher.isReady())
468                 {
469                     // Execute the broadcast request multicasting dispatcher. 05/21/2017, Bing Li
470                     super.execute(this.multicastBroadcastRequestDispatcher);
471                 }
472                 // Enqueue the broadcast request into the multicasting dispatcher. 05/21/2017, Bing Li
473                 this.multicastBroadcastRequestDispatcher.enqueue(helloWorldRequest);
474                 break;
475
476             case ClusterMessageType.HELLO_WORLD_ANYCAST_REQUEST:
477                 System.out.println("<" + ClusterChild.CLUSTER().getChildIP() + ":" + ClusterChild.CLUSTER()
478                     .getChildPort() + "> HELLO_WORLD_ANYCAST_REQUEST received @" +
479                     Calendar.getInstance().getTime());
480                 // Check whether the anycast request dispatcher is ready. 02/02/2016, Bing Li
481                 if (!this.processHelloWorldAnycastRequestDispatcher.isReady())
482                 {
483                     // Execute the anycast request dispatcher. 02/02/2016, Bing Li
484                     super.execute(this.processHelloWorldAnycastRequestDispatcher);
485                 }
486                 // Enqueue the notification into those anycast request dispatchers. The notifications are queued
487                 // and processed asynchronously with a proper synchronization. 11/27/2014, Bing Li
488                 this.processHelloWorldAnycastRequestDispatcher.enqueue((HelloWorldAnycastRequest)
489                 message.getMessage());
490                 break;
491
492             case ClusterMessageType.HELLO_WORLD_UNICAST_REQUEST:
493                 System.out.println("<" + ClusterChild.CLUSTER().getChildIP() + ":" + ClusterChild.CLUSTER()
494                     .getChildPort() + "> HELLO_WORLD_UNICAST_REQUEST received @" +
495                     Calendar.getInstance().getTime());
496                 // Check whether the unicast request dispatcher is ready. 02/02/2016, Bing Li
497                 if (!this.processHelloWorldUnicastRequestDispatcher.isReady())
498                 {
499                     // Execute the unicast request dispatcher. 02/02/2016, Bing Li
500                     super.execute(this.processHelloWorldUnicastRequestDispatcher);
501                 }
502                 // Enqueue the notification into those unicast request dispatchers. The notifications are queued
503                 // and processed asynchronously with a proper synchronization. 11/27/2014, Bing Li
504                 this.processHelloWorldUnicastRequestDispatcher.enqueue((HelloWorldUnicastRequest)
505                     message.getMessage());
506                 break;
507
508             case ClusterMessageType.SHUTDOWN_CHILDREN_BROADCAST_NOTIFICATION:
509                 System.out.println("<" + ClusterChild.CLUSTER().getChildIP() + ":" + ClusterChild.CLUSTER()
510                     .getChildPort() + "> SHUTDOWN_CHILDREN_BROADCAST_NOTIFICATION received @" +
511                     Calendar.getInstance().getTime());
512                 // Check whether the bound broadcast notification dispatcher is ready. 02/02/2016, Bing Li
513                 if (!this.processShutdownBroadcastNotificationDispatcher.isReady())
514                 {
515                     // Execute the bound broadcast notification dispatcher. 02/02/2016, Bing Li
516                     super.execute(this.processShutdownBroadcastNotificationDispatcher);
517                 }
518                 // Enqueue the notification into those bound broadcast notification dispatchers. The notifications
519                 // are queued and processed asynchronously with a proper synchronization. 11/27/2014, Bing Li
520                 this.processShutdownBroadcastNotificationDispatcher.enqueue
521                     ((ShutdownChildrenBroadcastNotification)message.getMessage());
522                 break;
523         }
524     }
525 }
```

List 7.40 The code of ChildDispatcher.java

This is long code, right? Do not scare since it follows our programming methodology

exactly. The efficiency to program it still rapidly if you keep the knowledge, learn APIs and perceive the patterns.

Line 35 declares one instance, rootIPNotificationDisposer, of ClusterMessageDisposer. This is the first time you encounter such a class when programming the server side using the pattern of SD. Basically, the class, ClusterMessageDisposer, is an implementation of the interface, MessageBindable.

The object, rootIPNotificationDisposer, is used to dispose the incoming notification. When a notification is consumed, it is suggested to dispose it to collect the resource. We have done that many times in previous examples. The first time about the issue happens when we discuss the code of ChatRegistryThread.java in List 5.15. At Line 48, the incoming request and response are disposed by invoking the method, disposeMessage(), inherited from the parent, RequestQueue. Similarly, incoming notifications are also needed to dispose in the API of NotificationQueue. You can check the code of AddPartnerThread.java of List 5.33. At Line 36, the notification is disposed after it is processed. If you review all of the instances of RequestQueue and NotificationQueue, you must notice that each of those threads in the pattern of DWC disposes requests/responses and notifications after they are consumed.

You must feel weird why we cannot do that in the same way. Why should we program a disposer that is not enclosed in the threads but independent of them? To answer the questions, we need to keep reading the code. I will answer you soon.

Line 39~41 declares an instance, processRootIPBroadcastNotificationDispatcher, of BoundNotificationDispatcher. We mention a new pattern, BND, just a couple of minutes ago (Table 7.27). The API needs to be programmed in the pattern. It looks familiar to some extent although it is new. In previous chapters, we learn the pattern of ND. There must be some relations between ND and BND? Yes, you are right. Besides the similarity between their names, they are made up with almost the same components, including the notification, the thread and the thread creator. However, BND has one additional component, the binder, which is implemented as the message disposer, which is not included in ND. In the case, the component is identical to the instance of ClusterMessageDisposer, i.e., rootIPNotificationDisposer. In Line 129, the instance is used to initialize processRootIPBroadcastNotificationDispatcher.

Likewise, Line 45~47 declares another instance, broadcastRootIPNotificationDispatcher, of BoundNotificationDispatcher. This is another usage for the new pattern of BND. Compared with the one you are familiar with, ND, the pattern has one additional component, ClusterMessageDisposer, as well. If you move forward a little bit to Line 147, you should feel surprised that the instance, rootIPNotificationDisposer, of ClusterMessageDisposer declared in Line 35 is also one parameter of the constructor of BoundNotificationDispatcher. Both broadcastRootIPNotificationDispatcher and processRootIPBroadcastNotificationDispatcher share the same message disposer! I am sure you will ask the question why it is necessary to do so.

To elaborate that clearly, we need to comprehend what the two BNDs do at the server side of the cluster child. The one, processRootIPBroadcastNotificationDispatcher, is responsible for keeping the received IP address of the cluster root. In contrast, another

one, broadcastRootIPNotificationDispatcher, is designed to forward the received notification of RootIPAddressBroadcastNotification to the children of the cluster child. In other words, when the notification of RootIPAddressBroadcastNotification is received by the cluster child, it prepares two concurrency mechanisms to process the notification asynchronously. That is a reasonable solution. It is ridiculous to do that synchronously since both of the tasks done by the two BNDs respectively are regarded as potential heavyweight ones. If processing synchronously, the overall performance of the cluster must be lowered. Moreover, since both of them need to process the same message of RootIPAddressBroadcastNotification asynchronously during the procedure, it might get conflicts if one of them disposes the notification without notifying others. It indicates that a synchronization mechanism should be designed for both of them. Therefore, if either of them finishes its task, it has to notify its counterpart. The message cannot be disposed unless both of them have finished their tasks, i.e., keeping the IP address of the cluster root and forwarding the notification to its children. Fortunately, the goal is fulfilled by the shared message disposer of ClusterMessageDisposer. That is also why the pattern's name has a prefix, Bound. It represents that both of concurrency dispatchers are synchronized for one particular issue although they run concurrently in most time.

More important, our case in the chapter is a contrived one to present you how to program clusters with GreatFree. So the collaboration between threads are simple. However, the issue is critical to real world systems. In practice, the synchronization must be different. Developers can program the binder to implement specific synchronization. In brief, the BND pattern provides developers with the approach to keep synchronous for concurrency mechanisms at the server side. This is an essential design for a computing system.

So now you must have understood the new pattern of BND.

Line 53 declares an instance, helloNotificationDisposer, of ClusterMessageDisposer. The message disposer is prepared for the later declared BNDs, i.e., processHelloBroadcastNotificationDispatcher and broadcastHelloWorldNotificationDispatcher. The design is identical to the one in Line 35, which is declared for processRootIPBroadcastNotificationDispatcher and broadcastRootIPNotificationDispatcher.

Line 57~59 declares an instance, processHelloBroadcastNotificationDispatcher, of BoundNotificationDispatcher. It shares the message disposer, helloNotificationDisposer (Line 53), with the later declared instance, broadcastHelloWorldNotificationDispatcher (Line 63~65), of BoundNotitificationDispatcher, to dispose notifications synchronously.

Line 63~65 declares an instance, broadcastHelloWorldNotificationDispatcher, of BoundNotificationDispatcher. It shared the message disposer, helloNotificationDisposer (Line 53), with the previous declared instance, processHelloBroadcastNotificationDispatcher, to dispose notifications synchronously.

Line 69~70 declares an instance, processHelloWorldAnycastNotificationDispatcher, of NotificationDispatcher. This is a classic ND, right? So I do not need to spend much time on it. However, what is the ND used for? Why do we need to design such a ND

here? The notification in this case is one anycasting message in the type of HelloWorldAnycastNotification, which is shown in List 7.9. However, when the multicasting notification is received, the approach to process it is different from what we have discussed in Line 35~63. At least, the notification is not forwarded. Actually, the topic should be resolved. When an anycasting notification is received, that means the cluster child receives the notification. According to the concept of anycasting, it indicates the goal of anycasting is achieved. So it is not necessary to forward the notification somewhere else. Thus, a classic ND is sufficient to handle the notification.

Similarly, Line 74~75 declares an instance, processHelloWorldUnicastNotificationDispatcher, of NotificationDispatche for notification unicasting. Again, it is not necessary to forward the received notification to the children of the cluster child.

Line 78 declares an instance, helloRequestDispatcher, of ClusterMessageDisposer. It is prepared for later concurrency code as well. But the change here is that the disposer is designed to dispose multicasting requests rather than notifications.

Line 80~82 declares an instance, processHelloWorldBroadcastRequestDispatcher, of BoundRequestDispatcher. You encounter a new pattern, BRD, again. Since it needs to process multicasting requests, we have to employ such a new pattern. Similar to the pattern of RD, BRD has some overlaps in their components, such as the request, the response, the thread and the thread creator. However, there are two differences between them. Besides the four components, RD has the component of the output message stream (OutMessageStream), which sends the generated response back to the request sender. But we do not find it in BRD, which has no such a component. Why? It is due to the fact that received requests are forward by the parent, which might not be the root. Thus, there is no a direct TCP connection between the root and the child. So the output message stream does not exist between them either. Thus, the component is missed in the new pattern of BRD.

Line 84~86 declares an instance, multicastBroadcastRequestDispatcher, of BoundNotificationDispatcher. It is responsible for forwarding the multicasting request to its children. Since both processHelloWorldBroadcastRequestDispatcher and multicastBroadcastRequestDispatcher needs to consume the same multicasting request. They have to keep synchronous at the moment when disposing the message such that we cannot use typical RD and ND.

Line 90~91 declares an instance, processHelloWorldAnycastRequestDispatcher, of NotificationDispatcher again for the anycasting request, HelloWorldAnycastRequest. The same as processing anycasting notifications, no forwarding is required. So the ND is enough for the task. However, you can still find some new phenomenon in the case. In the previous chapters, the ND is utilized to process notifications only. However, now the ND has to process requests. Can we employ the RD for that? No, in this case, you have to program with ND instead of RD. As I discuss previously, in a cluster, usually, there is no direct TCP connection between the cluster root and the cluster child. So the output message stream does not exist either. However, the component of the output message stream is required for a RD. Because of that, the only pattern that can be used here is ND. Inside the thread of ND, the cluster child has

to construct the TCP connection with the root directly before sending the response to it. That is why a cluster child needs to have the IP address of the cluster root, which is declared at Line 37 of ClusterChild in List 7.35.

Line 95~96 declares an instance, processHelloWorldUnicastRequestDispatcher, of NotificationDispatcher. It needs to process the unicasting request, HelloWorldUnicastRequest. For the same reason mentioned above, no forwarding is needed in the case. Meanwhile, the pattern of ND and the relevant APIs are the unique choice in the case.

Line 101~104 declares an instance, processShutdownBroadcastNotificationDispatcher, of NotificationDispatcher. The ND is used to broadcast the notification, ShutdownChildrenBroadcastNotification, to its children such that the entire cluster can be shut down. However, one question is raised again here. In Line 35~63, we have programmed code to process incoming broadcasting notifications. Well, in those cases, we prepare for BNDs to do that. But why in the case do we need to program with ND only? This is due to the particular message, ShutdownChildrenBroadcastNotification. When a cluster child receives the notification, it has to shut it down immediately. However, because a cluster child has the responsibility to forward the notification, it has to do that before it is shut down. So the two tasks have to be accomplished sequentially, forwarding first and then shutting the cluster child. For that, the pattern of BND cannot be employed.

Line 109~319 defines the constructor of ChildDispatcher. Inside it, besides the new patterns of BND and BRD, you must be familiar with the parameters for those initializations. So we focus on how to initialize the new patterns.

Line 112 initializes the parent class, ServerDispatcher, of ChildDispatcher.

Line 117 initializes the instance, rootIPNotificationDisposer, of ClusterMessageDisposer. No parameters are needed for that.

Line 122~136 initializes the instance, processRootIPBroadcastNotificationDispatcher. This is the new pattern, BND, you have to deal with. Fortunately, you do not need to be scared about the constructor parameters of the dispatcher, BoundNotificationDispatcher. You can review Table 5.9 about the constructor parameters of NotificationDispatcher. You must be surprised to that. The two dispatchers has only one difference. The one of BoundNotificationDispatcher has one additional parameter, Binder, which should implement the interface, MessageBindable. In the case, that parameter is the instance, rootIPNotificationDisposer, of ClusterMessageDisposer.

Line 140~154 initializes the instance, broadcastRootIPNotificationDispatcher. The additional parameter is rootIPNotificationDisposer, which is shared with processRootIPBroadcastNotificationDispatcher. Other parameters are identical to those specified in Table 5.9.

Line 158 initializes the instance, helloNotificationDisposer, which is shared by processHelloBroadcastNotificationDispatcher and broadcastHelloWorldNotificationDispatcher, to dispose incoming notifications

synchronously.

Line 162~176 initializes the instance, processHelloBroadcastNotificationDispatcher, which follows the pattern of BND.

Line 180~194 initializes the instance, broadcastHelloWorldNotificationDispatcher, which follows the pattern of BND.

Line 198~210 initializes the instance, processHelloWorldAnycastNotificationDispatcher, which follows the classic pattern of ND to dispatch and process anycasting notifications of the type, HelloWorldAnycastNotification.

Line 214~226 initializes the instance, processHelloWorldUnicastNotificationDispatcher, which follows the classic pattern of ND to dispatch unicasting notification of the type, HelloWorldUnicastNotification.

Line 230 initializes the message disposer, helloRequestDisposer.

Line 233~247 initializes the instance, processHelloWorldBroadcastRequestDispatcher, to dispatch and process broadcasting request, HelloWorldBroadcastRequest, which follows the pattern of BRD. I emphasize the additional parameter, helloRequestDisposer, again.

Line 250~264 initializes the instance, multicastBroadcastRequestDispatcher, to forward the broadcasting request, HelloWorldBroadcastRequest, to its children. The API should be programmed following the pattern of BND. Additionally, it needs to share the same binder, helloRequestDisposer, with processHelloWorldBroadcastRequestDispatcher.

Line 268~280 initializes the instance, processHelloWorldAnycastRequestDispatcher, to dispatch and process the anycasting request, HelloWorldAnycastRequest. This is a classic pattern of ND.

Line 283~295 initializes the instance, processHelloWorldUnicastRequestDispatcher, to dispatch and process the unicasting request, HelloWorldUnicastRequest. This is a classic pattern of ND as well.

Line 299~312 initializes the instance, processShutdownBroadcastNotificationDispatcher, to dispatch, process and forward the broadcasting request, ShutdownChildrenBroadcastNotification. To do the above jobs sequentially, the API of NotificationDispatcher is adopted.

Line 318~346 defines the method, shutdown(), to dispose all of the APIs in the dispatcher, ChildDispatcher.

Line 352~525 defines the method of consume(), which overrides the method of its parent, ServerDispatcher. All of the incoming messages should be dispatched within the method.

Line 359~386 dispatches the broadcasting notification, RootIPAddressBroadcastNotification. The code here is programmed in the pattern of SD, which is repeated many times. A little bit different from the SDs of the previous chapters, in the case statement, two dispatchers, BNDs, emerge to consume the notification.

Line 389~415 dispatches the broadcasting notification, HelloWorldBroadcastNotification. The code is placed in the exactly same pattern of SD. Again, two dispatchers, BNDs, are exhibited here to take the two tasks, i.e., processing notifications and forwarding notifications, respectively.

Line 418~432 dispatches the anycasting notification, HelloWorldAnycastNotification, to the ND, processHelloWorldAnycastNotificationDispatcher.

Line 434~448 dispatches the unicasting notification, HelloWorldUnicastNotification, to the ND, processHelloWorldUnicastNotificationDispatcher.

Line 450~474 dispatches, processes and forwards the broadcasting request, HelloWorldBroadcastRequest, through queuing messages into the corresponding dispatchers. Now two dispatchers, BRD and BND, are employed.

Line 476~490 dispatches the anycasting request, HelloWorldAnycastRequest, to the NotificationDispatcher, processHelloWorldAnycastRequestDispatcher.

Line 492~506 dispatches the unicasting request, HelloWorldUnicastRequest, to the NotificationDispatcher, processHelloWorldUnicastRequestDispatcher.

Line 508~522 dispatches, processes and forward the broadcasting notification, ShutdownChildrenBroadcastNotification. Different from Line 359~386 and Line 389~415, the lines deal with broadcasting notifications sequentially.

OK! I spend much time explaining the long code to you. This is the primary mechanism for the server side of the cluster child. Although the computing environment is updated from the one-to-one model, i.e., C/S or P2P, to the one-to-many one, the code, i.e., APIs and patterns, does not change a lot. However, I have to claim that the cluster programming is really different from C/S and P2P. Even as an application developer, you have to learn new APIs and patterns. Fortunately, the number of them is small and the programming procedure is still simplified.

3.6 BND: Bound Notification Dispatcher

The pattern, BND (Bound Notification Dispatcher), is an extension to the classic one, ND. To synchronize the threads managed by multiple dispatchers, which are the instances of BoundNotificationDispatcher or BoundRequestDispatcher, all of the threads need to share one additional class, a binder, which implements the interface, MessageBindable<ServerMessage>. If any one of the threads finish its task asynchronously, it is required to call the method, bind(), of the binder to achieve the goal of synchronization between them. The final synchronous task is accomplished until all of the threads invoke the method, bind().

You can get some examples of BND from the SD of the cluster child. In the specific case, multiple threads need to work concurrently once if one particular broadcasting message is received. Although those tasks are concurrent, they have to synchronize when disposing the shared message. To achieve this goal, the pattern, BND, is employed.

Since BND is an extension to ND, they have similar components, including the notification, the thread and the thread creator. Additionally, BND has one new component, called the binder, which is the origin of the pattern's name. For convenience, the components of BND are renamed as BN(Broadcasting Notification), i.e., the notification, BNQ (Bound Notification Queue), i.e., the thread, and BNQC (Bound Notification Queue Creator), i.e., the thread creator.

3.6.1 BND to Process the Broadcasting Notification: RootIPAddressBroadcastNotification

When the broadcasting notification, RootIPAddressBroadcastNotification, is received, the IP address of the cluster root should be retained by the cluster child. Likewise, it is mandatory to forward the notification to its children to keep broadcasting. The two tasks should be performed concurrently at the server side of the cluster child. However, since we need to dispose the message after the tasks are accomplished, the threads to take the tasks must be synchronized before doing that. Table 7.28 illustrates the pattern of BND.

Pattern: BND (Bound Notification Dispatcher)		
Class	BoundNotificationDispatcher<RootIPAddressBroadcastNotification, ClusterMessageDisposer<RootIPAddressBroadcastNotification>, RootIPAddressBroadcastNotificationThread, RootIPAddressBroadcastNotificationThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(RootIPAddressBroadcastNotification notification)	
Enclosed Pattern	BN (Broadcasting Notification)	RootIPAddressBroadcastNotification
	Binder	ClusterMessageDisposer<RootIPAddressBroadcastNotification>
	BNQ (Bound Notification Queue)	RootIPAddressBroadcastNotificationThread
	BNQC (Bound Notification Queue Creator)	RootIPAddressBroadcastNotificationThreadCreator
Employed API	ServerMessage	
	MulticastMessage	
	MessageBindable<ServerMessage>	
	ClusterMessageDisposer<Message extends MulticastMessage>	
	BoundNotificationQueue<Notification extends ServerMessage, Binder extends MessageBindable<Notification>>	
	BoundNotificationThreadCreatable<Notification extends ServerMessage, Binder extends MessageBindable<Notification>, NotificationThread extends BoundNotificationQueue<Notification, Binder>>	

Table 7.28 The pattern of BND to process RootIPAddressBroadcastNotification

The notification, RootIPAddressBroadcastNotification, for the BND is shown in List 7.4.

The binder, ClusterMesssageDisposer<Message extends MulticastMessage>, is supported by GreatFree, not an application code developers should implement. So if you are interested in the underlying code, you can navigate the code. Here, I just skip it.

The thread, RootIPAddressBroadcastNotificationThread, derives from BoundNotificationQueue. The code is shown in List 7.41. If you read the code, you must be surprised to its structure. Actually, it follows the pattern of DWC exactly. That means your programming experiences can be reused for cluster programming. Table 7.29 demonstrates the pattern of DWC for the thread.

```
1   package com.greatfree.cluster.child;
2
3   import com.greatfree.cluster.message.ClusterMessageDisposer;
4   import com.greatfree.cluster.message.RootIPAddressBroadcastNotification;
5   import com.greatfree.concurrency.BoundNotificationQueue;
6   import com.greatfree.testing.data.ServerConfig;
7
8   /*
9    * The thread processes the instances of RootIPAddressBroadcastNotification from the root. Since the
10   * notification is shared by the one that needs to forward it, both of them are derived from
11   * BoundNotificationQueue for synchronization. 11/26/2014, Bing Li
12   */
13
14  // Created: 05/20/2017, Bing Li
15  public class RootIPAddressBroadcastNotificationThread extends BoundNotificationQueue
16      <RootIPAddressBroadcastNotification, ClusterMessageDisposer<RootIPAddressBroadcastNotification>>
17  {
18      /*
19       * Initialize the thread. 11/26/2014, Bing Li
20       *
21       * In the constructor, the parameters are explained as follows.
22       *
23       *      int taskSize: the size of the notification queue.
24       *
25       *      String dispatcherKey: the key of the dispatcher which manages the thread. It is the thread key that
26       *          shares the notification in the binder below.
27       *
28       *      ClusterMessageDisposer<RootIPAddressBroadcastNotification> binder: the binder that
29       *          synchronizes the relevant threads that share the notification and disposes the notification after
30       *          the sharing is ended.
31       *
32       */
33      public RootIPAddressBroadcastNotificationThread(int taskSize, String dispatcherKey,
34          ClusterMessageDisposer<RootIPAddressBroadcastNotification> binder)
35      {
36          super(taskSize, dispatcherKey, binder);
37      }
38
39      /*
40       * The thread to process notifications asynchronously. 11/26/2014, Bing Li
41       */
42      @Override
43      public void run()
44      {
45          // Declare a notification instance of RootIPAddressBroadcastNotification. 11/26/2014, Bing Li
46          RootIPAddressBroadcastNotification notification;
47          // The thread always runs until it is shutdown by the BoundNotificationDispatcher. 11/26/2014, Bing Li
48          while (!this.isShutdown())
49          {
50              // Check whether the notification queue is empty. 11/26/2014, Bing Li
51              while (!this.isEmpty())
52              {
53                  try
54                  {
55                      // Dequeue the notification. 11/26/2014, Bing Li
56                      notification = this.getNotification();
57                      // Keep the root IP address. 05/20/2017, Bing Li
58                      ClusterChild.CLUSTER().setRootIP(notification.getRootAddress());
59                      // Notify the binder that the notification is consumed by the thread. 11/26/2014, Bing Li
60                      this.bind(super.getDispatcherKey(), notification);
61                  }
62                  catch (InterruptedException e)
63                  {
64                      e.printStackTrace();
65                  }
66              }
```

```
67                  try
68                  {
69                      // Wait for a moment after all of the existing notifications are processed. 11/26/2014, Bing Li
70                      this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
71                  }
72                  catch (InterruptedException e)
73                  {
74                      e.printStackTrace();
75                  }
76              }
77          }
78      }
```

List 7.41 The code of RootIPAddressBroadcastNotificationThread.java

Pattern: DWC (Double While Concurrency)		
Class	RootIPAddressBroadcastNotificationThread	
Singleton	No	
Method	void bind(String threadKey, RootIPAddressBroadcastNotification notification)	
	void enqueue(RootIPAddressBroadcastNotification notification)	
	RootIPAddressBroadcastNotification getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	RootIPAddressBroadcastNotification
	Notification Disposer	ClusterMessageDisposer< RootIPAddressBroadcastNotification>
Employed API	BoundNotificationQueue<Notification extends ServerMessage, Binder extends MessageBindable<Notification>>	
	ClusterMessageDisposer<Message extends MulticastMessage>	
	MulticastMessage	

Table 7.29 The pattern of DWC to Retain the IP address of the Cluster Root

Let us take a look at the code in List 7.41. First of all, according to Line 15~16, the thread derives from BoundNotificationQueue rather than NotificationQueue. This difference of the new parent is that it contains the binder to keep designated synchronization.

In Line 33~37, the constructor of the thread for BND is different from the previous ones for ND. In addition to the parameter of the task size (taskSize), two more parameters are included in the constructor, including the key that represents the thread, dispatcherKey, and the binder in the type of ClusterMessageDisposer<RootIPAddressBroadcastNotification>. The key is used as the parameter to notify other relevant threads that the task within the thread is done. Since the binder is shared by other relevant threads, when the method, bind(), is invoked with the key and the notification, the synchronization can be attained.

Line 43~77 defines the method of run() to process the concurrent task. However, since a synchronization is required, there must be some differences. The same thing happens at Line 58, which keeps the root IP address of the cluster child concurrently. In contrast, the different thing happens at Line 60, in which the method of bind() is called. The invocation aims to perform the synchronization.

Anyway, the change of the thread is small and easy to be grasped. So I still describe the code in the pattern of DWC.

The fourth component of the BND is the creator of the thread, i.e., RootIPAddressBroadcastNotificationThreadCreator.java, as shown in List 7.42.

```
1    package com.greatfree.cluster.child;
```

```
2
3   import com.greatfree.cluster.message.ClusterMessageDisposer;
4   import com.greatfree.cluster.message.RootIPAddressBroadcastNotification;
5   import com.greatfree.concurrency.BoundNotificationThreadCreatable;
6
7   /*
8    * This is an implementation of the interface BoundNotificationThreadCreatable to create the
9    * instance of RootIPAddressBroadcastNotificationThread inside the pool,
10   * BoundNotificationDispatcher. 11/26/2014, Bing Li
11   */
12
13  // Created: 05/20/2017, Bing Li
14  public class RootIPAddressBroadcastNotificationThreadCreator implements
15      BoundNotificationThreadCreatable<RootIPAddressBroadcastNotification, ClusterMessageDisposer
16      <RootIPAddressBroadcastNotification>, RootIPAddressBroadcastNotificationThread>
17  {
18      @Override
19      public RootIPAddressBroadcastNotificationThread createNotificationThreadInstance(int taskSize, String
20          dispatcherKey, ClusterMessageDisposer<RootIPAddressBroadcastNotification> binder)
21      {
22          return new RootIPAddressBroadcastNotificationThread(taskSize, dispatcherKey, binder);
23      }
24  }
```

List 7.42 The code of RootIPAddressBroadcastNotificationThreadCreator.java

The creator implements the interface, BoundNotificationThreadCreatable. Since the constructor of the thread is updated, the creator has to adapt to that as well. So that is easy to understand.

3.6.2 BND to Forward the Broadcasting Notification, RootIPAddressBroadcastNotification

Since you have learned the pattern of BND in the above section, Section 3.6.1, we can save much time if we program any other BND. Now let us implement how to forward the notification, RootIPAddressBroadcastNotification, to its children. Table 7.29 shows the pattern.

Pattern: BND (Bound Notification Dispatcher)		
Class	BoundNotificationDispatcher<RootIPAddressBroadcastNotification, ClusterMessageDisposer<RootIPAddressBroadcastNotification>, BroadcastRootIPAddressNotificationThread, BroadcastRootIPAddressNotificationThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(RootIPAddressBroadcastNotification notification)	
Enclosed Pattern	Notification	RootIPAddressBroadcastNotification
	Binder	ClusterMessageDisposer<RootIPAddressBroadcastNotification>
	BNQ (Bound Notification Queue)	BroadcastRootIPAddressNotificationThread
	BNQC (Bound Notification Queue Creator)	BroadcastRootIPAddressNotificationThreadCreator
Employed API	ServerMessage	
	MulticastMessage	
	MessageBindable<ServerMessage>	
	ClusterMessageDisposer<Message extends MulticastMessage>	
	BoundNotificationQueue<Notification extends ServerMessage, Binder extends MessageBindable<Notification>>	
	BoundNotificationThreadCreatable<Notification extends ServerMessage, Binder extends MessageBindable<Notification>, NotificationThread extends BoundNotificationQueue<Notification, Binder>>	

Table 7.30 The pattern of BND to forward the broadcasting notification to its children

The two components, the broadcasting notification, RootIPAddressBroadcastNotification, and the message disposer,

ClusterMessageDisposer, do not need to be presented.

The thread to forward the notification, BroadcastRootIPAddressNotificationThread, is shown in List 7.43. I do not think I need to explain the code in detail again. You are suggested to review the code of List 7.41. It needs to emphasize that both of the threads need to work asynchronously and synchronously. The concurrent task of the thread is performed at Line 62~63, which invokes the method, broadcastRootIP(), to forward the broadcasting notification to its children. Well, the sequential task is done at Line 65. It notifies another thread, RootIPAddressBroadcastNotificationThread, which shares the notification, that the notification can be disposed. Table 7.31 demonstrates the pattern of DWC for the thread.

Pattern: DWC (Double While Concurrency)		
Class	BroadcastRootIPAddressNotificationThread	
Singleton	No	
Method	void bind(String threadKey, RootIPAddressBroadcastNotification notification)	
	void enqueue(RootIPAddressBroadcastNotification notification)	
	RootIPAddressBroadcastNotification getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	RootIPAddressBroadcastNotification
	Notification Disposer	ClusterMessageDisposer<RootIPAddressBroadcastNotification>
Employed API	BoundNotificationQueue<Notification extends ServerMessage, Binder extends MessageBindable<Notification>>	
	ClusterMessageDisposer<Message extends MulticastMessage>	
	MulticastMessage	

Table 7.31 The pattern of DWC to forward the IP address of the cluster root to its children

```
1    package com.greatfree.cluster.child;
2
3    import java.io.IOException;
4
5    import com.greatfree.cluster.ClusterConfig;
6    import com.greatfree.cluster.message.ClusterMessageDisposer;
7    import com.greatfree.cluster.message.RootIPAddressBroadcastNotification;
8    import com.greatfree.concurrency.BoundNotificationQueue;
9    import com.greatfree.testing.data.ServerConfig;
10
11   /*
12    * The thread processes the notification of RootIPAddressBroadcastNotification concurrently. Since it shares
13    * the notifications with other threads, i.e., BroadcastRootIPAddressNotificationThread, it must have the
14    * mechanism of synchronization. Therefore, it derives the BoundNotificationQueue rather than
15    * NotificationQueue. 11/27/2014, Bing Li
16    */
17
18   // Created: 05/20/2017, Bing Li
19   public class BroadcastRootIPAddressNotificationThread extends BoundNotificationQueue
20       <RootIPAddressBroadcastNotification, ClusterMessageDisposer<RootIPAddressBroadcastNotification>>
21   {
22       /*
23        * Initialize the thread. 11/27/2014, Bing Li
24        *
25        * In the constructor, the parameters are explained as follows.
26        *
27        *       int taskSize: the size of the notification queue.
28        *
29        *       String dispatcherKey: the key of the dispatcher which manages the thread. It is the thread key that
30        *           shares the notification in the binder below.
31        *
32        *       ClusterMessageDisposer<RootIPAddressBroadcastNotification> disposer: the disposer that
33        *           synchronizes the relevant threads that share the notification and disposes the notification after
34        *           the sharing is ended.
35        *
36        */
37       public BroadcastRootIPAddressNotificationThread(int taskSize, String dispatcherKey,
```

```
38            ClusterMessageDisposer<RootIPAddressBroadcastNotification> binder)
39        {
40            super(taskSize, dispatcherKey, binder);
41        }
42
43        /*
44         * The thread to broadcast notifications asynchronously. 11/27/2014, Bing Li
45         */
46        @Override
47        public void run()
48        {
49            // Declare a notification instance of RootIPAddressBroadcastNotification. 11/27/2014, Bing Li
50            RootIPAddressBroadcastNotification notification;
51            // The thread always runs until it is shutdown by the BoundNotificationDispatcher. 11/27/2014, Bing Li
52            while (!this.isShutdown())
53            {
54                // Check whether the notification queue is empty. 11/27/2014, Bing Li
55                while (!this.isEmpty())
56                {
57                    try
58                    {
59                        // Dequeue the notification. 11/27/2014, Bing Li
60                        notification = this.getNotification();
61                        // Disseminate the notification of HelloWorldNotification among children. 11/27/2014, Bing Li
62                        ClusterChild.CLUSTER().broadcastRootIP(notification,
63                            ClusterConfig.SUB_BRANCH_COUNT);
64                        // Notify the binder that the notification is consumed by the thread. 11/27/2014, Bing Li
65                        this.bind(super.getDispatcherKey(), notification);
66                    }
67                    catch (InterruptedException | InstantiationException | IllegalAccessException | IOException e)
68                    {
69                        e.printStackTrace();
70                    }
71                }
72                try
73                {
74                    // Wait for a moment after all of the existing notifications are processed. 11/26/2014, Bing Li
75                    this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
76                }
77                catch (InterruptedException e)
78                {
79                    e.printStackTrace();
80                }
81            }
82        }
83    }
```

List 7.43 The code of BroadcastRootIPAddressNotificationThread.java

The thread creator, BroadcastRootIPAddressNotificationThreadCreator.java, is shown in List 7.44.

```
1     package com.greatfree.cluster.child;
2
3     import com.greatfree.cluster.message.ClusterMessageDisposer;s
4     import com.greatfree.cluster.message.RootIPAddressBroadcastNotification;
5     import com.greatfree.concurrency.BoundNotificationThreadCreatable;
6
7     // Created: 05/20/2017, Bing Li
8     public class BroadcastRootIPAddressNotificationThreadCreator implements
9         BoundNotificationThreadCreatable<RootIPAddressBroadcastNotification, ClusterMessageDisposer
10        <RootIPAddressBroadcastNotification>, BroadcastRootIPAddressNotificationThread>
11    {
12        @Override
13        public BroadcastRootIPAddressNotificationThread createNotificationThreadInstance(int taskSize, String
14            dispatcherKey, ClusterMessageDisposer<RootIPAddressBroadcastNotification> binder)
15        {
16            return new BroadcastRootIPAddressNotificationThread(taskSize, dispatcherKey, binder);
17        }
18    }
```

List 7.44 The code of BroadcastRootIPAddressNotificationThreadCreator.java

3.6.3 BND to Process the Broadcasting Notification: HelloWorldBroadcastNotification

The BND for the broadcasting notification, HelloWorldBroadcastNotification, is illustrated in Table 7.32. As a contrived notification, the relevant BND is programmed in the exactly same structure as the one for RootIPAddressBroadcastNotification, as discussed in Section 3.6.1.

Pattern: BND (Bound Notification Dispatcher)		
Class	BoundNotificationDispatcher<HelloWorldBroadcastNotification, ClusterMessageDisposer< HelloWorldBroadcastNotification>, HelloWorldBroadcastNotificationThread, HelloWorldBroadcastNotificationThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(HelloWorldBroadcastNotification notification)	
Enclosed Pattern	BN (Broadcasting Notification)	HelloWorldBroadcastNotification
	Binder	ClusterMessageDisposer<HelloWorldBroadcastNotification>
	BNQ (Bound Notification Queue)	HelloWorldBroadcastNotificationThread
	BNQC (Bound Notification Queue Creator)	HelloWorldBroadcastNotificationThreadCreator
Employed API	ServerMessage	
	MulticastMessage	
	MessageBindable<ServerMessage>	
	ClusterMessageDisposer<Message extends MulticastMessage>	
	BoundNotificationQueue<Notification extends ServerMessage, Binder extends MessageBindable<Notification>>	
	BoundNotificationThreadCreatable<Notification extends ServerMessage, Binder extends MessageBindable<Notification>, NotificationThread extends BoundNotificationQueue<Notification, Binder>>	

Table 7.32 The pattern of BND to process the broadcasting notification, HelloWorldBroadcastNotification

The first component of the notification, HelloWorldBroadcastNotification, is shown in List 7.7.

The message disposer, ClusterMessageDisposer<HelloWorldBroadcastNotification>, is supported by GreatFree. You do not need program it.

The thread, HelloWorldBroadcastNotificationThread, is shown in List 7.45. Immediately after receiving the broadcasting notification, HelloWorldBroadcastNotification, the message is displayed on the screen in this contrived case at Line 59~60. In practice, it should process the message by invoking relevant method. Usually, it must be a heavyweight task. So the concurrency pattern DWC is necessary. After the task is accomplished, it needs to call the method, bind(), at Line 62 to notify other threads that share the same message that the message can be disposed. Table 7.33 illustrates the pattern of DWC.

```
1    package com.greatfree.cluster.child;
2
3    import com.greatfree.cluster.message.ClusterMessageDisposer;
4    import com.greatfree.cluster.message.HelloWorldBroadcastNotification;
5    import com.greatfree.concurrency.BoundNotificationQueue;
6    import com.greatfree.testing.data.ServerConfig;
7
8    /*
9     * The thread processes the instances of HelloWorldNotification from the root. Since the notification is shared
10    * by the one that needs to forward it, both of them are derived from BoundNotificationQueue for
11    * synchronization. 11/26/2014, Bing Li
```

```
12      */
13
14      // Created: 05/15/2017, Bing Li
15      public class HelloWorldBroadcastNotificationThread extends BoundNotificationQueue
16          <HelloWorldBroadcastNotification, ClusterMessageDisposer<HelloWorldBroadcastNotification>>
17      {
18          /*
19           * Initialize the thread. 11/26/2014, Bing Li
20           *
21           * In the constructor, the parameters are explained as follows.
22           *
23           *       int taskSize: the size of the notification queue.
24           *
25           *       String dispatcherKey: the key of the dispatcher which manages the thread. It is the thread key that
26           *           shares the notification in the binder below.
27           *
28           *       ClusterMessageDisposer<HelloWorldBroadcastNotification> binder: the binder that synchronizes the
29           *           relevant threads that share the notification and disposes the notification after the sharing is
30           *           ended.
31           *
32           */
33          public HelloWorldBroadcastNotificationThread(int taskSize, String dispatcherKey,
34              ClusterMessageDisposer<HelloWorldBroadcastNotification> binder)
35          {
36              super(binder, dispatcherKey);
37          }
38
39          /*
40           * The thread to process notifications asynchronously. 11/26/2014, Bing Li
41           */
43          @Override
44          public void run()
45          {
46              // Declare a notification instance of HelloWorldNotification. 11/26/2014, Bing Li
47              HelloWorldBroadcastNotification notification;
48              // The thread always runs until it is shutdown by the BoundNotificationDispatcher. 11/26/2014, Bing Li
49              while (!this.isShutdown())
50              {
51                  // Check whether the notification queue is empty. 11/26/2014, Bing Li
52                  while (!this.isEmpty())
53                  {
54                      try
55                      {
56                          // Dequeue the notification. 11/26/2014, Bing Li
57                          notification = this.getNotification();
58                          // Just present the received message. 06/16/2017, Bing Li
59                          System.out.println("HelloWorldBroadcastNotificationThread-BroadcastNotification received: "
60                              + notification.getHelloWorld().getHelloWorld());
61                          // Notify the binder that the notification is consumed by the thread. 11/26/2014, Bing Li
62                          this.bind(super.getDispatcherKey(), notification);
63                      }
64                      catch (InterruptedException e)
65                      {
66                          e.printStackTrace();
67                      }
68                  }
69                  try
70                  {
71                      // Wait for a moment after all of the existing notifications are processed. 11/26/2014, Bing Li
72                      this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
73                  }
74                  catch (InterruptedException e)
75                  {
76                      e.printStackTrace();
77                  }
78              }
79          }
80      }
```

List 7.45 The code of HelloWorldBroadcastNotificationThread.java

The last component of the BND pattern is HelloWorldBroadcastNotificationThreadCreator.java, which is demonstrated in List

7.46.

```
1    package com.greatfree.cluster.child;
2
3    import com.greatfree.cluster.message.ClusterMessageDisposer;
4    import com.greatfree.cluster.message.HelloWorldBroadcastNotification;
5    import com.greatfree.concurrency.BoundNotificationThreadCreatable;
6
7    /*
8     * This is an implementation of the interface BoundNotificationThreadCreatable to create the instance of
9     * BroadcastNotificationThread inside the pool, BoundNotificationDispatcher. 11/26/2014, Bing Li
10    */
11
12   // Created: 05/15/2017, Bing Li
13   public class HelloWorldBroadcastNotificationThreadCreator implements
14       BoundNotificationThreadCreatable<HelloWorldBroadcastNotification, ClusterMessageDisposer
15       <HelloWorldBroadcastNotification>, HelloWorldBroadcastNotificationThread>
16   {
17       @Override
18       public HelloWorldBroadcastNotificationThread createNotificationThreadInstance(int taskSize, String
19           dispatcherKey, ClusterMessageDisposer<HelloWorldBroadcastNotification> binder)
20       {
21           return new HelloWorldBroadcastNotificationThread(taskSize, dispatcherKey, binder);
22       }
23   }
```

List 7.46 The code of HelloWorldBroadcastNotificatinThreadCreator.java

Pattern: DWC (Double While Concurrency)		
Class	HelloWorldBroadcastNotificationThread	
Singleton	No	
Method	void bind(String threadKey, HelloWorldBroadcastNotification notification)	
	void enqueue(HelloWorldBroadcastNotification notification)	
	HelloWorldBroadcastNotification getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	HelloWorldBroadcastNotification
	Notification Disposer	ClusterMessageDisposer<HelloWorldBroadcastNotification>
Employed API	BoundNotificationQueue<Notification extends ServerMessage, Binder extends MessageBindable<Notification>>	
	ClusterMessageDisposer<Message extends MulticastMessage>	
	MulticastMessage	

Table 7.33 The pattern of DWC to process the broadcasting notification concurrently and notify other threads for synchronization

3.6.4 BND to Forward the Broadcasting Notification: HelloWorldBroadcastNotification

When the broadcasting notification, HelloWorldBroadcastNotification, is received, another heavyweight task is to forward it to its children. After that is done, the thread, BroadcastHelloWorldNotificationThread (List 7.47), needs to notify the other one, HelloWorldBroadcastNotificationThread (List 7.45), that the message can be disposed. Table 7.34 illustrates the pattern of BND for the forwarding.

The code of BroadcastHelloWorldNotificationThread is shown in List 7.47. At Line 65~66, the thread invokes the method, broadcastNotify(), of ClusterChild (List 7.35), to forward the received notification to its children. At Line 73, it invokes the method, bind(), to notify the thread, HelloWorldBroadcastNotificationThread, that the message can be disposed.

```
1    package com.greatfree.cluster.child;
2
```

```
3   import java.io.IOException;
4
5   import com.greatfree.cluster.ClusterConfig;
6   import com.greatfree.cluster.message.ClusterMessageDisposer;
7   import com.greatfree.cluster.message.HelloWorldBroadcastNotification;
8   import com.greatfree.concurrency.BoundNotificationQueue;
9   import com.greatfree.testing.data.ServerConfig;
10
11  /*
12   * The thread processes the notification of HelloWorldNotification concurrently. Since it shares the notifications
13   * with other threads, i.e., HelloWorldNotificationThread, it must have the mechanism of synchronization.
14   * Therefore, it derives the BoundNotificationQueue rather than NotificationQueue. 11/27/2014, Bing Li
15   */
16
17  // Created: 05/16/2017, Bing Li
18  public class BroadcastHelloWorldNotificationThread extends BoundNotificationQueue
19      <HelloWorldBroadcastNotification, ClusterMessageDisposer<HelloWorldBroadcastNotification>>
20  {
21      /*
22       * Initialize the thread. 11/27/2014, Bing Li
23       *
24       * In the constructor, the parameters are explained as follows.
25       *
26       *      int taskSize: the size of the notification queue.
27       *
28       *      String dispatcherKey: the key of the dispatcher which manages the thread. It is the thread key that
29       *          shares the notification in the binder below.
30       *
31       *      ClusterMessageDisposer<HelloWorldBroadcastNotification> disposer: the disposer that
32       *          synchronizes the relevant threads that share the notification and disposes the notification after
33       *          the sharing is ended.
34       *
35       */
36      public BroadcastHelloWorldNotificationThread(int taskSize, String dispatcherKey,
37          ClusterMessageDisposer<HelloWorldBroadcastNotification> binder)
38      {
39          super(taskSize, dispatcherKey, binder);
40      }
41
42      /*
43       * The thread to broadcast notifications asynchronously. 11/27/2014, Bing Li
45       */
46      @Override
47      public void run()
48      {
49          // Declare a notification instance of HelloWorldNotification. 11/27/2014, Bing Li
50          HelloWorldBroadcastNotification notification;
51          // The thread always runs until it is shutdown by the BoundNotificationDispatcher. 11/27/2014, Bing Li
52          while (!this.isShutdown())
53          {
54              // Check whether the notification queue is empty. 11/27/2014, Bing Li
55              while (!this.isEmpty())
56              {
57                  try
58                  {
59                      // Dequeue the notification. 11/27/2014, Bing Li
60                      notification = this.getNotification();
61                      try
62                      {
63                          // Disseminate the notification of HelloWorldNotification among
64                          // children. 11/27/2014, Bing Li
65                          ClusterChild.CLUSTER().broadcastNotify(notification,
66                              ClusterConfig.SUB_BRANCH_COUNT);
67                      }
68                      catch (InstantiationException | IllegalAccessException | IOException e)
69                      {
70                          e.printStackTrace();
71                      }
72                      // Notify the binder that the notification is consumed by the thread. 11/27/2014, Bing Li
73                      this.bind(super.getDispatcherKey(), notification);
74                  }
75                  catch (InterruptedException e)
76                  {
77                      e.printStackTrace();
78                  }
79              }
80              try
```

```
81                {
82                    // Wait for a moment after all of the existing notifications are processed. 11/26/2014, Bing Li
83                    this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
84                }
85                catch (InterruptedException e)
86                {
87                    e.printStackTrace();
88                }
89            }
90        }
91    }
```

List 7.47 The code of BroadcastHelloWorldNotificationThread.java

Pattern: BND (Bound Notification Dispatcher)		
Class	BoundNotificationDispatcher<HelloWorldBroadcastNotification, ClusterMessageDisposer<HelloWorldBroadcastNotification>, BroadcastHelloWorldNotificationThread, BroadcastHelloWorldNotificationThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(HelloWorldBroadcastNotification notification)	
Enclosed Pattern	BN (Broadcasting Notification)	HelloWorldBroadcastNotification
	Binder	ClusterMessageDisposer<HelloWorldBroadcastNotification>
	BNQ (Bound Notification Queue)	BroadcastHelloWorldNotificationThread
	BNQC (Bound Notification Queue Creator)	BroadcastHelloWorldNotificationThreadCreator
Employed API	ServerMessage	
	MulticastMessage	
	MessageBindable<ServerMessage>	
	ClusterMessageDisposer<Message extends MulticastMessage>	
	BoundNotificationQueue<Notification extends ServerMessage, Binder extends MessageBindable<Notification>>	
	BoundNotificationThreadCreatable<Notification extends ServerMessage, Binder extends MessageBindable<Notification>, NotificationThread extends BoundNotificationQueue<Notification, Binder>>	

Table 7.34 The pattern of BND to forward the broadcasting notification

The pattern of DWC for the thread is summarized in Table 7.35.

Pattern: DWC (Double While Concurrency)		
Class	BroadcastHelloWorldNotificationThread	
Singleton	No	
Method	void bind(String threadKey, HelloWorldBroadcastNotification notification)	
	void enqueue(HelloWorldBroadcastNotification notification)	
	HelloWorldBroadcastNotification getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	HelloWorldBroadcastNotification
	Notification Disposer	ClusterMessageDisposer<HelloWorldBroadcastNotification>
Employed API	BoundNotificationQueue<Notification extends ServerMessage, Binder extends MessageBindable<Notification>>	
	ClusterMessageDisposer<Message extends MulticastMessage>	
	MulticastMessage	

Table 7.35 The pattern of DWC to forward the broadcasting notification and notify others for synchronization

The last component, i.e., the thread creator, is shown in List 7.48.

```
1    package com.greatfree.cluster.child;
2
3    import com.greatfree.cluster.message.ClusterMessageDisposer;
4    import com.greatfree.cluster.message.HelloWorldBroadcastNotification;
5    import com.greatfree.concurrency.BoundNotificationThreadCreatable;
6
7    /*
```

```
8       * The creator aims to generate the instance of BroadcastHelloWorldNotificationThread for the
9       * BoundNotificationDispatcher so as to schedule the notification as tasks concurrently. 11/27/2014, Bing Li
10      */
11
12     // Created: 05/16/2017, Bing Li
13     public class BroadcastHelloWorldNotificationThreadCreator implements
14         BoundNotificationThreadCreatable<HelloWorldBroadcastNotification, ClusterMessageDisposer
15         <HelloWorldBroadcastNotification>, BroadcastHelloWorldNotificationThread>
16     {
17         @Override
18         public BroadcastHelloWorldNotificationThread createNotificationThreadInstance(int taskSize, String
19             dispatcherKey, ClusterMessageDisposer<HelloWorldBroadcastNotification> binder)
20         {
21             return new BroadcastHelloWorldNotificationThread(taskSize, dispatcherKey, binder);
22         }
23     }
```

List 7.48 The code of BroadcastHelloWorldNotificationThreadCreator.java

3.7 BRD: Bound Request Dispatcher

Usually, there are no direct TCP connections between a cluster root and a cluster child. Thus, even though we need to reply the request to the cluster root at the cluster child, the dispatcher's interface and structure become different from the one, in which a client sends a request directly to a server. For the same reason, the pattern, BRD (Bound Request Dispatcher), looks more like the one of BND than that of RD, which we program a lot in the previous chapters.

Since the BRD needs to consider generating the response, the BRD is made up with five components, i.e., the request, the response, the request disposer, the thread and the thread creator.

Pattern: BRD (Bound Request Dispatcher)		
Class	BoundRequestDispatcher<HelloWorldBroadcastRequest, HelloWorldBroadcastResponse, ClusterMessageDisposer<HelloWorldBroadcastRequest>, HelloWorldBroadcastRequestThread, HelloWorldBroadcastRequestThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(HelloWorldBroadcastRequest request)	
Enclosed Pattern	Request	HelloWorldBroadcastRequest
	Response	HelloWorldBroadcastResponse
	Binder	ClusterMessageDisposer< HelloWorldBroadcastRequest >
	BRQ (Bound Request Queue)	HelloWorldBroadcastRequestThread
	BRQC (Bound Request Queue Creator)	HelloWorldBroadcastRequestThreadCreator
Employed API	ServerMessage	
	MulticastMessage	
	MessageBindable<ServerMessage>	
	ClusterMessageDisposer<Message extends MulticastMessage>	
	BoundRequestQueue<Request extends ServerMessage, Response extends ServerMessage, RequestBinder extends MessageBindable<Request>>	
	BoundRequestThreadCreatable<Request extends ServerMessage, Response extends ServerMessage, RequestBinder extends MessageBindable<Request>, RequestThread extends BoundRequestQueue<Request, Response, RequestBinder>>	

Table 7.36 The pattern of BRD to dispatch and process the broadcasting notification

3.7.1 BRD to Process the Broadcasting Request: HelloWorldBroadcastRequest

In our case between Line 80~82, one BRD is employed. When a broadcasting request, HelloWorldBroadcastRequest, is received, the same as the pattern of BND, BRD

needs to not only complete the concurrent task but also keep synchronous with the other thread that shares the same message before disposing it. Table 7.36 illustrates the pattern.

The request of the BRD is HelloWorldBroadcastRequest, which is shown in List 7.15.

The response of the BRD is HelloWorldBroadcastResponse, which is shown in List 7.16.

The request disposer is the same the one for broadcasting notifications. The only update is to replace notifications with the request. It is handled by Java Generics. So we do not need to spend time on that.

The thread, HelloWorldBroadcastRequestThread, should be explained for a little bit time. The code is presented in List 7.49. Before elaborating it, we just summarizes the pattern of DWC for it in Table 7.37.

```
1    package com.greatfree.cluster.child;
2
3    import java.io.IOException;
4
5    import com.greatfree.cluster.HelloWorld;
6    import com.greatfree.cluster.message.ClusterMessageDisposer;
7    import com.greatfree.cluster.message.HelloWorldBroadcastRequest;
8    import com.greatfree.cluster.message.HelloWorldBroadcastResponse;
9    import com.greatfree.concurrency.BoundRequestQueue;
10   import com.greatfree.testing.data.Constants;
11   import com.greatfree.testing.data.ServerConfig;
12   import com.greatfree.util.Tools;
13
14   /*
15    * The thread deals with the request and responds to the broadcast request initiator that sends the request.
16    * Since the request that contains the message must be broadcast to the local node's children, the disposal of
17    * the request must be controlled by the binder. 11/29/2014, Bing Li
18    */
19
20   // Created: 05/20/2017, Bing Li
21   public class HelloWorldBroadcastRequestThread extends BoundRequestQueue
22       <HelloWorldBroadcastRequest, HelloWorldBroadcastResponse, ClusterMessageDisposer
23       <HelloWorldBroadcastRequest>>
24   {
25       /*
26        * Initialize the thread. The binder is assigned at this point. 11/29/2014, Bing Li
27        */
28       public HelloWorldBroadcastRequestThread(int taskSize, String dispatcherKey, ClusterMessageDisposer
29           <HelloWorldBroadcastRequest> reqBinder)
30       {
31           super(taskSize, dispatcherKey, reqBinder);
32       }
33
34       /*
35        * Deal with the request and respond to the broadcast request initiator concurrently. After that, it must notify
36        * the binder that its task is accomplished. 11/29/2014, Bing Li
37        */
38       @Override
39       public void run()
40       {
41           // The instance of the received broadcast request. 11/29/2014, Bing Li
42           HelloWorldBroadcastRequest request;
43           // The instance of the response to be responded to the broadcast request initiator. 11/29/2014, Bing Li
44           HelloWorldBroadcastResponse response;
45           // The thread always runs until it is shutdown by the
46           // BoundBroadcastRequestDispatcher. 11/29/2014, Bing Li
47           while (!this.isShutdown())
48           {
49               // Check whether the request queue is empty. 11/29/2014, Bing Li
50               while (!this.isEmpty())
51               {
```

```
52                  try
53                  {
54                      // Dequeue the request. 11/29/2014, Bing Li
55                      request = this.getRequest();
56                      // Print the query on the screen. 06/17/2017, Bing Li
57                      System.out.println("Query: " + request.getHelloWorld().getHelloWorld());
58                      // Create the response. The unique key is required so that the initiator is able to estimate the
59                      // count of responses. That is different from the anycast response, which does not need to
60                      // have the key. 11/29/2014, Bing Li
61                      response = new HelloWorldBroadcastResponse(new HelloWorld
62                          (Constants.BROADCAST_RESPONSE + request.getHelloWorld().getHelloWorld() +
63                          " id: " + request.getKey()), Tools.generateUniqueKey(), request.getCollaboratorKey());
64                      // Respond the root of the broadcast requesting. 11/29/2014, Bing Li
65                      ClusterChild.CLUSTER().respondToRoot(response);
66                      // Notify the binder that the thread task on the request has done. 11/29/2014, Bing Li
67                      this.bind(super.getDispatcherKey(), request);
68                      // Dispose the response. 11/29/2014, Bing Li
69                      this.disposeResponse(response);
70                  }
71                  catch (InterruptedException | IOException e)
72                  {
73                      e.printStackTrace();
74                  }
75              }
76              try
77              {
78                  // Wait for a moment after all of the existing requests are processed. 11/29/2014, Bing Li
79                  this.holdOn(ServerConfig.REQUEST_THREAD_WAIT_TIME);
80              }
81              catch (InterruptedException e)
82              {
83                  e.printStackTrace();
84              }
85          }
86      }
87  }
```

List 7.49 The code of HelloWorldBroadcastRequestThread.java

Pattern: DWC (Double While Concurrency)		
Class	HelloWorldBroadcastRequestThread	
Singleton	No	
Method	void bind(String threadKey, HelloWorldBroadcastRequest request)	
	void enqueue(HelloWorldBroadcastRequest request)	
	HelloWorldBroadcastRequest getRequest()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
	void disposeResponse(HelloWorldBroadcastResponse response)	
Enclosed Pattern	Request	HelloWorldBroadcastRequest
	Response	HelloWorldBroadcastResponse
	Request Disposer	ClusterMessageDisposer< HelloWorldBroadcastRequest>
Employed API	BoundRequestQueue<Request extends ServerMessage, Response extends ServerMessage, RequestBinder extends MessageBindable<Request>>	
	ClusterMessageDisposer<Message extends MulticastMessage>	
	MulticastMessage	
	ClusterBroadcastRequest	
	ClusterBroadcastResponse	

Table 7.37 The pattern of DWC to process the broadcasting request and keep synchronous before disposing the request

According to Table 7.37, the thread has the same structure, DWC, as the ones for NotificationQueue, RequestQueue and even BoundNotificationQueue. Its parent class is BoundRequestQueue, which is derived at Line 21~23.

In Line 28~32, the same as the threads of BoundNotificationQueue, its constructor includes two additional parameters, the dispatcher key (dispatcherKey of String) and

the binder (reqBinder of ClusterMessageDisposer<HelloWorldBroadcastRequest>) in addition to the one for task size (taskSize of int).

Line 61~63 generates the response of HelloWorldBroadcastResponse. For the contrived sample, the response is created in a very simple way. You can update it with your practical code for your applications.

Line 65 sends the response to the cluster root through invoking the method, respondToRoot() of ClusterChild, which is defined at Line 196~199 in List 7.35. After the step, the concurrent task to process the request is attained.

Line 67 invokes the method, bind(), to notify the thread, BroadcastHelloWorldRequestThread in List 7.51 which shares the same request, that the request can be disposed.

Line 69 disposes the generated response after it is sent back to the root.

The last component of the BRD is HelloWorldBroadcastRequestThreadCreator, as shown in List 7.50. Although it implements the interface, BoundRequestThreadCreatable, whose name is different from BoundNotificationThreadCreatable for BND, the interface is identical.

```
1    package com.greatfree.cluster.child;
2
3    import com.greatfree.cluster.message.ClusterMessageDisposer;
4    import com.greatfree.cluster.message.HelloWorldBroadcastRequest;
5    import com.greatfree.cluster.message.HelloWorldBroadcastResponse;
6    import com.greatfree.concurrency.BoundRequestThreadCreatable;
7
8    /*
9     * The creator is responsible for initializing an instance of HelloWorldBroadcastRequestThread.
10    * It works with the BoundBroadcastRequestDispatcher to manage the searching process
11    * efficiently and concurrently. 11/29/2014, Bing Li
12    */
13
14   // Created: 05/20/2017, Bing Li
15   public class HelloWorldBroadcastRequestThreadCreator implements BoundRequestThreadCreatable
16       <HelloWorldBroadcastRequest, HelloWorldBroadcastResponse, ClusterMessageDisposer
17       <HelloWorldBroadcastRequest>, HelloWorldBroadcastRequestThread>
18   {
19       @Override
20       public HelloWorldBroadcastRequestThread createRequestThreadInstance(int taskSize, String
21           dispatcherKey, ClusterMessageDisposer<HelloWorldBroadcastRequest> reqBinder)
22       {
23           return new HelloWorldBroadcastRequestThread(taskSize, dispatcherKey, reqBinder);
24       }
25   }
```

List 7.50 The code of HelloWorldBroadcastRequestThreadCreator.java

3.7.2 BND to Forward the Broadcasting Request: HelloWorldBroadcastRequest

Now we need to talk about one instance of BND again. You must feel weird because of that. But as I indicated before, BRD is the same as BND in their overall structures. So you do not need to get confused for that. The BND we will discuss shares the broadcasting request, HelloWorldBroadcastRequest, with the BRD we just discuss in Section 3.7.1. Hence, it is necessary to keep synchronous before disposing the request.

As usual, Table 7.38 presents the pattern of the BND. Although the message, HelloWorldBroadcastRequest, is named as a request, it also inherits the parent class, ServerMessage. So it can be processed by BND.

Pattern: BND (Bound Notification Dispatcher)		
Class	BoundNotificationDispatcher<HelloWorldBroadcastRequest, ClusterMessageDisposer<HelloWorldBroadcastRequest>, BroadcastHelloWorldRequestThread, BroadcastHelloWorldRequestThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(HelloWorldBroadcastRequest notification)	
Enclosed Pattern	BN (Broadcasting Notification)	HelloWorldBroadcastRequest
	Binder	ClusterMessageDisposer< HelloWorldBroadcastRequest >
	BNQ (Bound Notification Queue)	BroadcastHelloWorldRequestThread
	BNQC (Bound Notification Queue Creator)	BroadcastHelloWorldRequestThreadCreator
Employed API	ServerMessage	
	MulticastMessage	
	MessageBindable<ServerMessage>	
	ClusterMessageDisposer<Message extends MulticastMessage>	
	BoundNotificationQueue<Notification extends ServerMessage, Binder extends MessageBindable<Notification>>	
	BoundNotificationThreadCreatable<Notification extends ServerMessage, Binder extends MessageBindable<Notification>, NotificationThread extends BoundNotificationQueue<Notification, Binder>>	

Table 7.38 The pattern of BND to forward the broadcasting request to its children and keep synchronous for disposing the request

We can skip the code of the request, HelloWorldBroadcastRequest (List 7.15), since it is discussed in Section 2.4.1.

The request disposer, ClusterMessageDisposer<HelloWorldBroadcastRequest>, can also be skipped.

The thread to forward the request is shown in List 7.51. The thread follows the pattern of DWC for sure. Table 7.39 summarizes the pattern. The structure of the code is identical to the ones of BNDs in Section 3.6. One superficial difference is that the notifications in those BNDs are replaced with the request, HelloWorldBroadcastRequest. However, since the thread, i.e., BoundNotificatinQueue, needs ServerMessage, which is the parent of HelloWorldBroadcastRequest, that is fine.

At Line 48~49, the request is forwarded to its children by calling the method, broadcastRead(), of ClusterChild. The method is defined in Line 222~226 of List 7.35.

```
1    package com.greatfree.cluster.child;
2
3    import java.io.IOException;
4
5    import com.greatfree.cluster.ClusterConfig;
6    import com.greatfree.cluster.message.ClusterMessageDisposer;
7    import com.greatfree.cluster.message.HelloWorldBroadcastRequest;
8    import com.greatfree.concurrency.BoundNotificationQueue;
9    import com.greatfree.testing.data.ServerConfig;
10
11   /*
12    * The thread broadcasts the instance of HelloWorldBroadcastRequest to the local
13    * node's children. 11/29/2014, Bing Li
```

```
14      */
15
16      // Created: 05/20/2017, Bing Li
17      public class BroadcastHelloWorldRequestThread extends BoundNotificationQueue
18          <HelloWorldBroadcastRequest, ClusterMessageDisposer<HelloWorldBroadcastRequest>>
19      {
20          /*
21           * Initialize the bound thread. 11/29/2014, Bing Li
22           */
23          public BroadcastHelloWorldRequestThread(int taskSize, String dispatcherKey, ClusterMessageDisposer
24              <HelloWorldBroadcastRequest> binder)
25          {
26              super(taskSize, dispatcherKey, binder);
27          }
28
29          /*
30           * Broadcast the request concurrently. 11/29/2014, Bing Li
31           */
32          @Override
33          public void run()
34          {
35              // The instance of the received broadcast request. 11/29/2014, Bing Li
36              HelloWorldBroadcastRequest request;
37              // The thread always runs until it is shutdown by the BoundNotificationDispatcher. 11/29/2014, Bing Li
38              while (!this.isShutdown())
39              {
40                  // Check whether the notification queue is empty. 11/29/2014, Bing Li
41                  while (!this.isEmpty())
42                  {
43                      try
44                      {
45                          // Dequeue the request. 11/29/2014, Bing Li
46                          request = this.getNotification();
47                          // Disseminate the broadcast request to the local node's children. 11/29/2014, Bing Li
48                          ClusterChild.CLUSTER().broadcastRead(request,
49                              ClusterConfig.SUB_BRANCH_COUNT);
50                          // Notify the binder that the thread's task on the request has done. 11/29/2014, Bing Li
51                          this.bind(super.getDispatcherKey(), request);
52                      }
53                      catch (InterruptedException | InstantiationException | IllegalAccessException | IOException e)
54                      {
55                          e.printStackTrace();
56                      }
57                  }
58                  try
59                  {
60                      // Wait for a moment after all of the existing requests are processed. 11/29/2014, Bing Li
61                      this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
62                  }
63                  catch (InterruptedException e)
64                  {
65                      e.printStackTrace();
66                  }
67              }
68          }
69      }
```

List 7.51 The code of BroadcastHelloWorldRequestThread.java

The thread creator, BroadcastHelloWorldRequestThreadCreator, is shown in List 7.52.

```
1       package com.greatfree.cluster.child;
2
3       import com.greatfree.cluster.message.ClusterMessageDisposer;
4       import com.greatfree.cluster.message.HelloWorldBroadcastRequest;
5       import com.greatfree.concurrency.BoundNotificationThreadCreatable;
6
7       /*
8        * The creator initialize the instance of the thread, TransmitBroadcastRequestThread. The
9        * BoundNotificationDispatcher can schedule tasks to the thread. 11/29/2014, Bing Li
10       */
11
```

```
12    // Created: 05/20/2017, Bing Li
13    public class BroadcastHelloWorldRequestThreadCreator implements BoundNotificationThreadCreatable
14        <HelloWorldBroadcastRequest, ClusterMessageDisposer<HelloWorldBroadcastRequest>,
15        BroadcastHelloWorldRequestThread>
16    {
17        @Override
18        public BroadcastHelloWorldRequestThread createNotificationThreadInstance(int taskSize, String
19            dispatcherKey, ClusterMessageDisposer<HelloWorldBroadcastRequest> binder)
20        {
21            return new BroadcastHelloWorldRequestThread(taskSize, dispatcherKey, binder);
22        }
23    }
```

List 7.52 The code of BroadcastHelloWorldRequestThreadCreator.java

Pattern: DWC (Double While Concurrency)		
Class	BroadcastHelloWorldRequestThread	
Singleton	No	
Method	void bind(String threadKey, HelloWorldBroadcastRequest notification)	
	void enqueue(HelloWorldBroadcastRequest notification)	
	HelloWorldBroadcastRequest getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	HelloWorldBroadcastRequest
	Notification Disposer	ClusterMessageDisposer<HelloWorldBroadcastRequest>
Employed API	BoundNotificationQueue<Notification extends ServerMessage, Binder extends MessageBindable<Notification>>	
	ClusterMessageDisposer<Message extends MulticastMessage>	
	MulticastMessage	

Table 7.39 The pattern of DWC to forward the broadcasting request and notify the thread, HelloWorldBroadcastRequestThread, for disposing the request

3.8 ND: Processing Anycasting & Unicasting

At the server side of the cluster child, it is unnecessary to forward anycasting and unicasting messages. It has been explained previously. For that, when receiving those messages, the pattern of ND can handle that concurrently. So this section contains nothing new. Its significance is to acknowledge you that GreatFree is a highly patterned programming environment.

3.8.1 ND for HelloWorldAnycastNotification

As an anycasting notification, HelloWorldAnycastNotification, the cluster child does not need to forward it but process it locally. The pattern is shown in Table 7.40.

The notification, HelloWorldAnycastNotification, is shown in List 7.9.

The thread is derived from the API you get used to, NotificationQueue. The code is shown in List 7.53. The primary task of the thread is to display the message on the screen (Line 28~29). In practice, you can replace the lines with your code in the real world.

```
1     package com.greatfree.cluster.child;
2
3     import com.greatfree.cluster.message.HelloWorldAnycastNotification;
4     import com.greatfree.concurrency.NotificationQueue;
5     import com.greatfree.testing.data.ServerConfig;
6
```

```
7       // Created: 05/19/2017, Bing Li
8       public class HelloWorldAnycastNotificationThread extends NotificationQueue
9           <HelloWorldAnycastNotification>
10      {
11          public HelloWorldAnycastNotificationThread(int taskSize)
12          {
13              super(taskSize);
14          }
15
16          @Override
17          public void run()
18          {
19              HelloWorldAnycastNotification notification;
20              while (!this.isShutdown())
21              {
22                  while (!this.isEmpty())
23                  {
24                      try
25                      {
26                          notification = this.getNotification();
27                          // Display the message on the screen. 06/17/2017, Bing Li
28                          System.out.println("HelloWorldAnycastNotificationThread-notification: " +
29                              notification.getHelloWorld().getHelloWorld());
30                          this.disposeMessage(notification);
31                      }
32                      catch (InterruptedException e)
33                      {
34                          e.printStackTrace();
35                      }
36                  }
37                  try
38                  {
39                      // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
40                      this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
41                  }
42                  catch (InterruptedException e)
43                  {
44                      e.printStackTrace();
45                  }
46              }
47          }
48      }
```

List 7.53 The code of HelloWorldAnycastNotificationThread.java

Pattern: ND (Notification Dispatcher)		
Class	NotificationDispatcher<HelloWorldAnycastNotification, HelloWorldAnycastNotificationThread, HelloWorldAnycastNotificationThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(HelloWorldAnycastNotification notification)	
Enclosed Pattern	Notification	HelloWorldAnycastNotification
	NQ (Notification Queue)	HelloWorldAnycastNotificationThread
	NQC (Notification Queue Creator)	HelloWorldAnycastNotificationThreadCreator
Employed API	ServerMessage	
	NotificationQueue<Notification extends ServerMessage>	
	NotificationThreadCreatable<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>>	

Table 7.40 The pattern of ND to dispatch the anycasting notification, HelloWorldAnycastNotification

To be clear, I exhibit the pattern of DWC for the thread in Table 7.41 for your reference.

The thread creator is named as HelloWorldAnycastNotificationThreadCreator, which is exhibited in List 7.54.

```
1       package com.greatfree.cluster.child;
```

```
2
3   import com.greatfree.cluster.message.HelloWorldAnycastNotification;
4   import com.greatfree.concurrency.NotificationThreadCreatable;
5
6   /*
7    * The HelloWorldAnycastNotificationThread creator generates such a thread when the current alive
8    * threads are busy. 05/21/2017, Bing Li
9    */
10
11  // Created: 05/19/2017, Bing Li
12  public class HelloWorldAnycastNotificationThreadCreator implements NotificationThreadCreatable
13      <HelloWorldAnycastNotification, HelloWorldAnycastNotificationThread>
14  {
15      @Override
16      public HelloWorldAnycastNotificationThread createNotificationThreadInstance(int taskSize)
17      {
18          return new HelloWorldAnycastNotificationThread(taskSize);
19      }
20  }
```

List 7.54 The code of HelloWorldAnycastNotificationThreadCreator.java

Pattern: DWC (Double While Concurrency)		
Class	HelloWorldAnycastNotificationThread	
Singleton	No	
Method	void dispose()	
	void enqueue(HelloWorldAnycastNotification notification)	
	HelloWorldAnycastNotification getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	HelloWorldAnycastNotification
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 7.41 The pattern of DWC to process the anycasting notification, HelloWorldAnycastNotification

3.8.2 ND for HelloWorldUnicastNotification

When a unicasting notification is received by a cluster child, it is unnecessary to forward the message to its children. What it should do on it is just the same as the typical manner of ND.

Table 7.42 illustrates the pattern of ND for the unicasting notification, HelloWorldUnicastNotification.

Pattern: ND (Notification Dispatcher)		
Class	NotificationDispatcher<HelloWorldUnicastNotification, HelloWorldUnicastNotificationThread, HelloWorldUnicastNotificationThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(HelloWorldUnicastNotification notification)	
Enclosed Pattern	Notification	HelloWorldUnicastNotification
	NQ (Notification Queue)	HelloWorldUnicastNotificationThread
	NQC (Notification Queue Creator)	HelloWorldUnicastNotificationThreadCreator
Employed API	ServerMessage	
	NotificationQueue<Notification extends ServerMessage>	
	NotificationThreadCreatable<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>>	

Table 7.42 The pattern of ND to dispatch the unicasting notification, HelloWorldUnicastNotification

The unicasting notification, HelloWorldUnicastNotification, is shown in List 7.11.

The thread, derived from NotificationQueue, is named as HelloWorldUnicastNotificationThread, which is shown in List 7.55. At Line 28~29 displays the message on the screen. Table 7.43 illustrates the pattern of DWC which the thread follows.

```
1   package com.greatfree.cluster.child;
2
3   import com.greatfree.cluster.message.HelloWorldUnicastNotification;
4   import com.greatfree.concurrency.NotificationQueue;
5   import com.greatfree.testing.data.ServerConfig;
6
7   // Created: 05/19/2017, Bing Li
8   public class HelloWorldUnicastNotificationThread extends NotificationQueue
9       <HelloWorldUnicastNotification>
10  {
11      public HelloWorldUnicastNotificationThread(int taskSize)
12      {
13          super(taskSize);
14      }
15
16      @Override
17      public void run()
18      {
19          HelloWorldUnicastNotification notification;
20          while (!this.isShutdown())
21          {
22              while (!this.isEmpty())
23              {
24                  try
25                  {
26                      notification = this.getNotification();
27                      // Display the message on the screen. 06/17/2017, Bing Li
28                      System.out.println("HelloWorldUnicastNotificationThread-notification: " +
29                          notification.getHelloWorld().getHelloWorld());
30                      this.disposeMessage(notification);
31                  }
32                  catch (InterruptedException e)
33                  {
34                      e.printStackTrace();
35                  }
36              }
37              try
38              {
39                  // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
40                  this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
41              }
42              catch (InterruptedException e)
43              {
44                  e.printStackTrace();
45              }
46          }
47      }
48  }
```

List 7.55 The code of HelloWorldUnicastNotificationThread.java

The thread creator, HelloWorldUnicastNotificationThreadCreator, is shown in List 7.56.

```
1   package com.greatfree.cluster.child;
2
3   import com.greatfree.cluster.message.HelloWorldUnicastNotification;
4   import com.greatfree.concurrency.NotificationThreadCreatable;
5
6   // Created: 05/19/2017, Bing Li
7   public class HelloWorldUnicastNotificationThreadCreator implements NotificationThreadCreatable
```

```
8           <HelloWorldUnicastNotification, HelloWorldUnicastNotificationThread>
9       {
10          @Override
11          public HelloWorldUnicastNotificationThread createNotificationThreadInstance(int taskSize)
12          {
13              return new HelloWorldUnicastNotificationThread(taskSize);
14          }
15      }
```

List 7.56 The code of HelloWorldUnicastNotificationThread.java

Pattern: DWC (Double While Concurrency)		
Class	HelloWorldUnicastNotificationThread	
Singleton	No	
Method	void dispose()	
	void enqueue(HelloWorldUnicastNotification notification)	
	HelloWorldUnicastNotification getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	HelloWorldUnicastNotification
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 7.43 The pattern of DWC to dispatch the unicasting notification, HelloWorldUnicastNotification

3.8.3 ND for HelloWorldAnycastRequest

We need to discuss another typical case using ND at the server side of the cluster child. That is, when an anycasting request is received by the cluster child, it is not necessary to forward the request to its children either. What it needs to do is to generate the response and reply to the cluster root.

Pattern: ND (Notification Dispatcher)		
Class	NotificationDispatcher<HelloWorldAnycastRequest, HelloWorldAnycastRequestThread, HelloWorldAnycastRequestThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(HelloWorldAnycastRequest notification)	
Enclosed Pattern	Notification	HelloWorldAnycastRequest
	NQ (Notification Queue)	HelloWorldAnycastRequestThread
	NQC (Notification Queue Creator)	HelloWorldAnycastRequestThreadCreator
Employed API	ServerMessage	
	NotificationQueue<Notification extends ServerMessage>	
	NotificationThreadCreatable<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>>	

Table 7.44 The pattern of ND to dispatch the anycasting request, HelloWorldAnycastRequest

In our case, the anycasting request, HelloWorldAnycastRequest, is received by the cluster child. You might wonder a request should be processed by RD rather than ND according to your experiences. However, the dispatcher we are currently employed is ND. In fact, we have discussed about the problem. Since there is no a direct TCP connection between the cluster root and the cluster child, it is impossible for the API, output message stream, to be the component to reply to the cluster root. So we have to employ ND instead of RD. Additionally, inside ND, we can reply to the cluster root with the instance, childPeer which is declared in Line 35 of ClusterChild in List 7.35.

Table 7.44 illustrates the pattern of ND for the anycasting request, HelloWorldAnycastRequest.

The anycasting request, HelloWorldAnycastRequest, is shown in List 7.18.

The thread, derived from NotificationQueue, HelloWorldAnycastRequestThread, is shown in List 7.57. When the request is received, the thread generates a response and replies it to the cluster root at Line 47. You must have invoked the method, which is respondToRoot of ClusterChild defined at Line 196~199 in List 7.35. It is invoked at Line 65 of HelloWorldBroadcastRequestThread.java in List 7.49. According to the code, you should learn that the response to a multicasting request is replied in a different way from what we program in the models of one-to-one interaction, i.e., C/S and P2P.

Table 7.45 illustrates the pattern of DWC the thread follows.

```
1    package com.greatfree.cluster.child;
2
3    import java.io.IOException;
4
5    import com.greatfree.cluster.HelloWorld;
6    import com.greatfree.cluster.message.HelloWorldAnycastRequest;
7    import com.greatfree.cluster.message.HelloWorldAnycastResponse;
8    import com.greatfree.concurrency.NotificationQueue;
9    import com.greatfree.testing.data.Constants;
10   import com.greatfree.testing.data.ServerConfig;
11   import com.greatfree.util.Tools;
12
13   /*
14    * Process the anycast request and reply to the root concurrently. 05/21/2017, Bing Li
15    */
16
17   // Created: 05/21/2017, Bing Li
18   public class HelloWorldAnycastRequestThread extends NotificationQueue<HelloWorldAnycastRequest>
19   {
20       public HelloWorldAnycastRequestThread(int taskSize)
21       {
22           super(taskSize);
23       }
24
25       @Override
26       public void run()
27       {
28           HelloWorldAnycastRequest request;
29           HelloWorldAnycastResponse response;
30           while (!this.isShutdown())
31           {
32               while (!this.isEmpty())
33               {
34                   try
35                   {
36                       // Dequeue the request. 11/29/2014, Bing Li
37                       request = this.getNotification();
38                       // Print the query on the screen. 06/17/2017, Bing Li
39                       System.out.println("Query: " + request.getHelloWorld().getHelloWorld());
40                       // Create the response. The unique key is required so that the initiator is able to estimate the
41                       // count of responses. That is different from the anycast response, which does not need to
42                       // have the key. 11/29/2014, Bing Li
43                       response = new HelloWorldAnycastResponse(new HelloWorld
44                           (Constants.ANYCAST_RESPONSE + request.getHelloWorld().getHelloWorld() + " id: "
45                           + request.getKey()), Tools.generateUniqueKey(), request.getCollaboratorKey());
46                       // Respond the root of the anycast requesting. 11/29/2014, Bing Li
47                       ClusterChild.CLUSTER().respondToRoot(response);
48                       this.disposeMessage(request);
49                   }
50                   catch (InterruptedException | IOException e)
51                   {
52                       e.printStackTrace();
```

```
53                  }
54              }
55              try
56              {
57                  // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
58                  this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
59              }
60              catch (InterruptedException e)
61              {
62                  e.printStackTrace();
63              }
64          }
65      }
66  }
```

List 7.57 The code of HelloWorldAnycastRequestThread.java

Pattern: DWC (Double While Concurrency)		
Class	HelloWorldAnycastRequestThread	
Singleton	No	
Method	void dispose()	
	void enqueue(HelloWorldAnycastRequest notification)	
	HelloWorldAnycastRequest getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	HelloWorldAnycastRequest
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 7.45 The pattern of DWC to process the anycasting request, HelloWorldAnycastRequest

The thread creator, HelloWorldAnycastRequestThreadCreator, is shown in List 7.58.

```
1   package com.greatfree.cluster.child;
2
3   import com.greatfree.cluster.message.HelloWorldAnycastRequest;
4   import com.greatfree.concurrency.NotificationThreadCreatable;
5
6   /*
7    * The HelloWorldAnycastNotificationThread creator generates such a thread when the current alive
8    * threads are busy. 05/21/2017, Bing Li
9    */
10
11  // Created: 05/21/2017, Bing Li
12  public class HelloWorldAnycastRequestThreadCreator implements NotificationThreadCreatable
13      <HelloWorldAnycastRequest, HelloWorldAnycastRequestThread>
14  {
15      @Override
16      public HelloWorldAnycastRequestThread createNotificationThreadInstance(int taskSize)
17      {
18          return new HelloWorldAnycastRequestThread(taskSize);
19      }
20  }
```

List 7.58 The code of HelloWorldAnycastRequestThreadCreator.java

3.8.4 ND for HelloWorldUnicastRequest

When a unicasting request is received, the code pattern on the cluster child is identical to that for an anycasting one. Table 7.46 presents the pattern of ND for the unicasting request, HelloWorldUnicastRequest.

The unicasting request, HelloWorldUnicastRequest, is shown in List 7.21.

The thread, HelloWorldUnicastRequestThread, is shown in List 7.59. The contrived code never does something complicated. That is normal, right? When the unicasting request is received, the thread just generates a response quickly at Line 43~45. After that, the response is sent back to the cluster root by invoking the method, respondToRoot() of ClusterChild again at Line 47. The method is defined at Line 196~199 in List 7.35. Table 7.47 demonstrates the DWC the thread follows.

Pattern: ND (Notification Dispatcher)		
Class	NotificationDispatcher<HelloWorldUnicastRequest, HelloWorldUnicastRequestThread, HelloWorldUnicastRequestThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(HelloWorldUnicastRequest notification)	
Enclosed Pattern	Notification	HelloWorldUnicastRequest
	NQ (Notification Queue)	HelloWorldUnicastRequestThread
	NQC (Notification Queue Creator)	HelloWorldUnicastRequestThreadCreator
Employed API	ServerMessage	
	NotificationQueue<Notification extends ServerMessage>	
	NotificationThreadCreatable<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>>	

Table 7.46 The pattern of ND to dispatch the unicasting request, HelloWorldUnicastRequest

```
1   package com.greatfree.cluster.child;
2
3   import java.io.IOException;
4
5   import com.greatfree.cluster.HelloWorld;
6   import com.greatfree.cluster.message.HelloWorldUnicastRequest;
7   import com.greatfree.cluster.message.HelloWorldUnicastResponse;
8   import com.greatfree.concurrency.NotificationQueue;
9   import com.greatfree.testing.data.Constants;
10  import com.greatfree.testing.data.ServerConfig;
11  import com.greatfree.util.Tools;
12
13  /*
14   * Process the unicast request and reply to the root concurrently. 05/21/2017, Bing Li
15   */
16
17  // Created: 05/21/2017, Bing Li
18  public class HelloWorldUnicastRequestThread extends NotificationQueue<HelloWorldUnicastRequest>
19  {
20      public HelloWorldUnicastRequestThread(int taskSize)
21      {
22          super(taskSize);
23      }
24
25      @Override
26      public void run()
27      {
28          HelloWorldUnicastRequest request;
29          HelloWorldUnicastResponse response;
30          while (!this.isShutdown())
31          {
32              while (!this.isEmpty())
33              {
34                  try
35                  {
36                      // Dequeue the request. 11/29/2014, Bing Li
37                      request = this.getNotification();
38                      // Print the query on the screen. 06/17/2017, Bing Li
39                      System.out.println("Query: " + request.getHelloWorld().getHelloWorld());
40                      // Create the response. The unique key is required so that the initiator is able to estimate the
41                      // count of responses. That is different from the anycast response, which does not need to
42                      // have the key. 11/29/2014, Bing Li
43                      response = new HelloWorldUnicastResponse(new HelloWorld
44                          (Constants.UNICAST_RESPONSE + request.getHelloWorld().getHelloWorld() + " id: "
45                          + request.getKey()), Tools.generateUniqueKey(), request.getCollaboratorKey());
```

```
46                      // Respond the root of the unicast requesting. 11/29/2014, Bing Li
47                      ClusterChild.CLUSTER().respondToRoot(response);
48                      this.disposeMessage(request);
49                  }
50                  catch (InterruptedException | IOException e)
51                  {
52                      e.printStackTrace();
53                  }
54              }
55              try
56              {
57                  // Wait for a moment after all of the existing notifications are processed. 01/20/2016, Bing Li
58                  this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
59              }
60              catch (InterruptedException e)
61              {
62                  e.printStackTrace();
63              }
64          }
65      }
66  }
```

List 7.59 The code of HelloWorldUnicastRequestThread.java

Pattern: DWC (Double While Concurrency)		
Class	HelloWorldUnicastRequestThread	
Singleton	No	
Method	void dispose()	
	void enqueue(HelloWorldUnicastRequest notification)	
	HelloWorldUnicastRequest getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	HelloWorldUnicastRequest
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 7.47 The pattern of DWC to process the unicasting request, HelloWorldUnicastRequest

The threat creator, HelloWorldUnicastRequestThreadCreator, is shown in List 7.60.

```
1   package com.greatfree.cluster.child;
2
3   import com.greatfree.cluster.message.HelloWorldUnicastRequest;
4   import com.greatfree.concurrency.NotificationThreadCreatable;
5
6   /*
7    * The HelloWorldUnicastRequestThread creator generates such a thread when the current alive
8    * threads are busy. 05/21/2017, Bing Li
9    */
10
11  // Created: 05/21/2017, Bing Li
12  public class HelloWorldUnicastRequestThreadCreator implements NotificationThreadCreatable
13      <HelloWorldUnicastRequest, HelloWorldUnicastRequestThread>
14  {
15      @Override
16      public HelloWorldUnicastRequestThread createNotificationThreadInstance(int taskSize)
17      {
18          return new HelloWorldUnicastRequestThread(taskSize);
19      }
20  }
```

List 7.60 The code of HelloWorldUnicastRequestThreadCreator.java

3.9 ND: Processing Broadcasting Notifications

We almost finish programming the contrived cluster. The last piece of code is written

in the pattern of ND. That is our old friend. The ND is designed to process a one-to-one notification rather than a multicasting notification. Although the notification, ShutdownChildrenBroadcastNotification, derives MulticastMessage, it is still originated from ServerMessage such that it can be processed by ND.

The notification intends to shut down all of the cluster children within the cluster. That means the notification should be broadcast to each of the child. We have talked about the design in Section 3.5. When the broadcasting notification is received, two tasks must be performed, i.e., shutting down the local child and forwarding the notification to its children. However, the task cannot be done concurrently. So although this is a notification of broadcasting, it is not processed in the way as others using BND in Section 3.6.

Table 7.48 illustrates the pattern of ND to dispatch the notification.

Pattern: ND (Notification Dispatcher)		
Class	NotificationDispatcher<ShutdownChildrenBroadcastNotification, ShutdownChildrenBroadcastNotificationThread, ShutdownChildrenBroadcastNotificationThreadCreator>	
Singleton	No	
Method	void dispose()	
	boolean isReady()	
	void enqueue(ShutdownChildrenBroadcastNotification notification)	
Enclosed Pattern	Notification	ShutdownChildrenBroadcastNotification
	NQ (Notification Queue)	ShutdownChildrenBroadcastNotificationThread
	NQC (Notification Queue Creator)	ShutdownChildrenBroadcastNotificationThreadCreator
Employed API	ServerMessage	
	NotificationQueue<Notification extends ServerMessage>	
	NotificationThreadCreatable<Notification extends ServerMessage, NotificationThread extends NotificationQueue<Notification>>	

Table 7.48 The pattern of ND to dispatch the broadcasting notification, ShutdownChildrenBroadcastNotification

The broadcasting notification, ShutdownChildrenBroadcastNotification, is shown in List 7.13.

The thread, ShutdownChildBroadcastNotificationThread, is shown in List 7.61. The code follows the pattern of DWC for sure again. Table 7.49 illustrates the pattern. Once if the notification is received, it tries to complete two tasks sequentially. Line 45~55 forwards the notification to its children by invoking the method, broadcastShutdownNotify(), of ClusterChild defined at Line 271~275 in List 7.2. The task should done earlier than the one to shut down the local child. No question, right? Line 57~66 shuts the child down by calling the method, stop(), of ClusterChild, which is defined at Line 103~125 in List 7.2.

```
1    package com.greatfree.cluster.child;
2
3    import java.io.IOException;
4
5    import com.greatfree.cluster.ClusterConfig;
6    import com.greatfree.cluster.message.ShutdownChildrenBroadcastNotification;
7    import com.greatfree.concurrency.NotificationQueue;
8    import com.greatfree.exceptions.RemoteReadException;
9    import com.greatfree.testing.data.ServerConfig;
10   import com.greatfree.util.ServerStatus;
11
12   /*
13    * The thread processes the instances of ShutdownChildrenBroadcastNotification from the coordinator. Since
14    * the notification is shared by the one that needs to forward it, both of them are derived from
15    * BoundNotificationQueue for synchronization. 11/26/2014, Bing Li
```

```
16      */
17
18      // Created: 05/19/2017, Bing Li
19      public class ShutdownChildrenBroadcastNotificationThread extends NotificationQueue
20          <ShutdownChildrenBroadcastNotification>
21      {
22          public ShutdownChildrenBroadcastNotificationThread(int taskSize)
23          {
24              super(taskSize);
25          }
26
27          /*
28           * The thread to process notifications asynchronously. 11/26/2014, Bing Li
29           */
30          @Override
31          public void run()
32          {
33              // Declare a notification instance of ShutdownChildrenBroadcastNotification. 11/26/2014, Bing Li
34              ShutdownChildrenBroadcastNotification notification;
35              // The thread always runs until it is shutdown by the BoundNotificationDispatcher. 11/26/2014, Bing Li
36              while (!this.isShutdown())
37              {
38                  // Check whether the notification queue is empty. 11/26/2014, Bing Li
39                  while (!this.isEmpty())
40                  {
41                      try
42                      {
43                          // Dequeue the notification. 11/26/2014, Bing Li
44                          notification = this.getNotification();
45                          try
46                          {
47                              // Disseminate the notification of ShutdownChildrenBroadcastNotification
48                              // among children. 11/27/2014, Bing Li
49                              ClusterChild.CLUSTER().broadcastShutdownNotify(notification,
50                                  ClusterConfig.SUB_BRANCH_COUNT);
51                          }
52                          catch (InstantiationException | IllegalAccessException e)
53                          {
54                              e.printStackTrace();
55                          }
56                          // Set the shutdown status. 05/20/2017, Bing Li
57                          ServerStatus.FREE().setShutdown();
58                          try
59                          {
60                              // Shutdown the child. 05/19/2017, Bing Li
61                              ClusterChild.CLUSTER().stop();
62                          }
63                          catch (ClassNotFoundException | RemoteReadException e)
64                          {
65                              e.printStackTrace();
66                          }
67                          this.disposeMessage(notification);
68                      }
69                      catch (InterruptedException | IOException e)
70                      {
71                          e.printStackTrace();
72                      }
73                  }
74                  try
75                  {
76                      // Wait for a moment after all of the existing notifications are processed. 11/26/2014, Bing Li
77                      this.holdOn(ServerConfig.NOTIFICATION_THREAD_WAIT_TIME);
78                  }
79                  catch (InterruptedException e)
80                  {
81                      e.printStackTrace();
82                  }
83              }
84          }
85      }
```

List 7.61 The code of ShutdownChildrenBroadcastNotificationThread.java

The thread creator, ShutdownChildrenBroadcastNotificationThreadCreator, is

presented in List 7.62.

```
1    package com.greatfree.cluster.child;
2
3    import com.greatfree.cluster.message.ShutdownChildrenBroadcastNotification;
4    import com.greatfree.concurrency.NotificationThreadCreatable;
5
6    /*
7     * This is an implementation of the interface BoundNotificationThreadCreatable to create the instance of
8     * BroadcastNotificationThread inside the pool, BoundNotificationDispatcher. 11/26/2014, Bing Li
9     */
10
11   // Created: 05/19/2017, Bing Li
12   public class ShutdownChildrenBroadcastNotificationThreadCreator implements
13       NotificationThreadCreatable<ShutdownChildrenBroadcastNotification,
14       ShutdownChildrenBroadcastNotificationThread>
15   {
16       @Override
17       public ShutdownChildrenBroadcastNotificationThread createNotificationThreadInstance(int taskSize)
18       {
19           return new ShutdownChildrenBroadcastNotificationThread(taskSize);
20       }
21   }
```

List 7.62 The code of ShutdownChildrenBroadcastNotificationThreadCreator.java

Pattern: DWC (Double While Concurrency)		
Class	ShutdownChildrenBroadcastNotificationThread	
Singleton	No	
Method	void dispose()	
	void enqueue(ShutdownChildrenBroadcastNotification notification)	
	ShutdownChildrenBroadcastNotification getNotification()	
	void holdOn(long waitTime)	
	boolean isShutdown()	
	boolean isEmpty()	
Enclosed Pattern	Notification	ShutdownChildrenBroadcastNotification
Employed API	NotificationQueue<Notification extends ServerMessage>	
	ServerMessage	

Table 7.49 The pattern of DWC to process the broadcasting notification, ShutdownChildrenBroadcastNotification

4. Testing

Let us test the system we have talked about a lot. Although it is not a real world system, you still can learn a lot about GreatFree. All of the topics in the chapter supper you to program a distributed system based on one-to-many interactions rather than one-to-one interactions. When the scale is large, the programming techniques are critical.

4.1 Constructing the Cluster

To test the system, I prepare four PCs again. I plan to I could set up as many children as possible. But actually I cannot. But until now, I believe the version is good. If you really detect some problems when the scale of your system becomes large, you can email me, bing.li@asu.edu.

To exhibit the system clearly, I create a cluster which contains seven computers, including one root and four children. It is not so convenient to present so many terminals in a book. I try to add some prompts on the screen. So I am confident that

we can verify that GreatFree cluster programming really works! Let us start!

Before starting the testing, you need to make sure that each Ant project must update the constant, CHAT_REGISTRY_ADDRESS, at Line 13 of ChatConfig.java in List 5.56, to the IP address where your chatting registry server is located.

The first step, let us start up the chatting registry server. Figure 7.1 illustrates that the server is started up.

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatRegistry — ssh — 80×40
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:1306 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1306 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:102299 (102.2 KB)  TX bytes:102299 (102.2 KB)

libing@freeweb:~/GreatFreeLabs/Clouds/ChatRegistry$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ChatRegistry/bu
ild

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/cl
asses
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/ja
r

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/Chat
Registry/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/j
ar/FreeWeb.jar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
```

Figure 7.1 The chatting registry server is started up

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
          collisions:0 txqueuelen:1000
          RX bytes:56215899 (56.2 MB)  TX bytes:4878146 (4.8 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:3497 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3497 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:285856 (285.8 KB)  TX bytes:285856 (285.8 KB)

libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8944
     [java] ===================
     [java]
     [java] The child is started ...
```

Figure 7.2 The child, 109.8944, is started up

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
          collisions:0 txqueuelen:1000
          RX bytes:56222193 (56.2 MB)  TX bytes:4887510 (4.8 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:3497 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3497 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:285856 (285.8 KB)  TX bytes:285856 (285.8 KB)

libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8945
     [java] ===================
     [java]
     [java] The child is started ...
```

Figure 7.3 The child, 109:8945, is started up

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
          collisions:0 txqueuelen:0
          RX bytes:215514 (215.5 KB)  TX bytes:215514 (215.5 KB)

wlan0     Link encap:Ethernet  HWaddr 58:94:6b:a5:05:a4
          inet addr:192.168.1.111  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::5a94:6bff:fea5:5a4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:92399 errors:0 dropped:0 overruns:0 frame:0
          TX packets:46159 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:57286094 (57.2 MB)  TX bytes:6891281 (6.8 MB)

libing@freescape:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.111:8944
     [java] ===================
     [java]
     [java] The child is started ...
```

Figure 7.4 The child, 111:8944, is started up

The second step, you need to start up the four children one by one. Actually, you can do that concurrently if your have enough hands. Figure 7.2~7.5 illustrates that the children are started up. You can see that each child has its own unique IP address, which is coordinated by the chatting registry server. Even though their IPs might be the same, their addresses are different from each other if their ports are taken into account. Figure 7.6 demonstrates the updates in the terminal of the chatting registry server.

To elaborate the testing clearly, each child is identified by their last three digit of IPs plus their ports. For example, according to Figure 7.2, the child is called 109:8944

since the last three digit of its IP is 192.168.1.109 and it port number is 8944. Using the rule, the other three children are named as 109:8945, 111:8944 and 107:8944.

```
imac — ssh — 85×40
        inet 192.168.1.107 netmask 0xffffff00 broadcast 192.168.1.255
        nd6 options=1<PERFORMNUD>
        media: autoselect
        status: active
fw0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 4078
        lladdr 00:26:b0:ff:fe:ed:f2:10
        nd6 options=1<PERFORMNUD>
        media: autoselect <full-duplex>
        status: inactive
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
        ether 0a:1e:df:df:91:0d
        media: autoselect
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ClusterChild/build/cl
asses

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ClusterChild/build/jar/FreeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.107:8944
     [java] ===================
     [java]
     [java] The child is started ...
```

Figure 7.5 The child, 107:8944, is started up

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatRegistry — ssh — 80×40
          TX packets:1306 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:102299 (102.2 KB)  TX bytes:102299 (102.2 KB)

libing@freeweb:~/GreatFreeLabs/Clouds/ChatRegistry$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ChatRegistry/bu
ild

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/cl
asses
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/ja
r

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/Chat
Registry/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/j
ar/FreeWeb.jar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:17 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:19 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:29 CST 2017
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:31 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:40 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:42 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:51 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:53 CST 2017
```

Figure 7.6 The updates on the chatting registry server after the six children are started

The third step, you need to start up the cluster root. Figure 7.7 demonstrates the terminal after the cluster root is started up. Meanwhile, Figure 7.8 demonstrates the updates on the chatting registry server. You notice that all of the terminals' of the cluster children are updated as well. The updates on the cluster children are illustrated

in Figure 7.9~7.12.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterRoot — ssh — 85×40
clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterRoot/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterRoot/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterRoot/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterRoot/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterRo
ot/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterRoot/build/jar/Fre
eWeb.jar

run:
     [java] Cluster root starting up ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.111:8945
     [java] ===================
     [java]
     [java] Cluster IPs = 192.168.1.109, 8944
     [java] Cluster IPs = 192.168.1.107, 8944
     [java] Cluster IPs = 192.168.1.109, 8945
     [java] Cluster IPs = 192.168.1.111, 8944
     [java] Cluster root started ...
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Broadcast a notification
     [java]     2) Unicast a notification
     [java]     3) Anycast a notification
     [java]     4) Broadcast a request
     [java]     5) Unicast a request
     [java]     6) Anycast a request
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
```

Figure 7.7 The cluster root is started up

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatRegistry — ssh — 80×40
libing@freeweb:~/GreatFreeLabs/Clouds/ChatRegistry$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ChatRegistry/bu
ild

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/cl
asses
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/ja
r

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/Chat
Registry/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/j
ar/FreeWeb.jar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:17 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:19 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:29 CST 2017
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:31 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:40 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:42 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:51 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:53 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:20:06 CST 2017
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] CLUSTER_IP_REQUEST received @Sun Jun 18 07:20:06 CST 2017
```

Figure 7.8 The updates on the chatting registry server after the cluster root is started up

I need to explain the procedure to set up the cluster. When each cluster child is started up, each of them registers on the chatting registry server. After that, the cluster root is started. It obtains all of the IP addresses of those cluster children from the chatting registry. You can see that the registry server receives a request, CLUSTER_IP_REQUEST. That indicates that the cluster root sends the request and gets the response. All of the IP address are displayed in Figure 7.7. To construct the cluster, the cluster root needs to make a broadcasting notification to all of the cluster

children about its IP address. The address is used to reply to the root when it performs request multicasting.

Till now the cluster is established. We can move forward to check whether the cluster really works together or not.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:3497 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3497 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:285856 (285.8 KB)  TX bytes:285856 (285.8 KB)

libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
```

Figure 7.9 The child, 109:8944, gets the IP address of the cluster root

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:3497 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3497 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:285856 (285.8 KB)  TX bytes:285856 (285.8 KB)

libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8945
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8945> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
```

Figure 7.10 The child, 109:8945, gets the IP address of the cluster root

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40

wlan0     Link encap:Ethernet  HWaddr 58:94:6b:a5:05:a4
          inet addr:192.168.1.111  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::5a94:6bff:fea5:5a4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:92399 errors:0 dropped:0 overruns:0 frame:0
          TX packets:46159 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:57286094 (57.2 MB)  TX bytes:6891281 (6.8 MB)

libing@freescape:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.111:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.111:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
```

Figure 7.11 The child, 111:8944, gets the IP address of the cluster root

```
imac — ssh — 85×40
        media: autoselect
        status: active
fw0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 4078
        lladdr 00:26:b0:ff:fe:ed:f2:10
        nd6 options=1<PERFORMNUD>
        media: autoselect <full-duplex>
        status: inactive
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
        ether 0a:1e:df:df:91:0d
        media: autoselect
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ClusterChild/build/cl
asses

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ClusterChild/build/jar/FreeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.107:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.107:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:24:34 PDT 2017
```

Figure 7.12 The child, 107:8944, gets the IP address of the cluster root

4.2 Performing the Notification Broadcasting

Figure 7.7 presents a simple menu to you. There are seven options for you to choose. Among them, six types of multicasting can be performed. Let us take a trial for the first one, “Broadcast a notification”.

After you select the first option, a submenu is displayed such that you can type the

message you want to broadcast to each of cluster children. Follow the menu and type the message through your keyboard. For example, you can type, “Hello, World!”. After that, each cluster child must receive the message.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterRoot — ssh — 85×40
     [java] ===================
     [java]
     [java] Cluster IPs = 192.168.1.109, 8944
     [java] Cluster IPs = 192.168.1.107, 8944
     [java] Cluster IPs = 192.168.1.109, 8945
     [java] Cluster IPs = 192.168.1.111, 8944
     [java] Cluster root started ...
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Broadcast a notification
     [java]     2) Unicast a notification
     [java]     3) Anycast a notification
     [java]     4) Broadcast a request
     [java]     5) Unicast a request
     [java]     6) Anycast a request
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
1
     [java] Your choice: 1
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 1
     [java] Please type your message:
Hello, World!
     [java] You notification is broadcast!
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 7.13 The procedure to perform the notification broadcasting at the cluster root

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:3497 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3497 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:285856 (285.8 KB)  TX bytes:285856 (285.8 KB)

libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.109:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
```

Figure 7.14 The child, 109:8944, receives the broadcasting notification, “Hello, World!”

Figure 7.13 shows the procedure that you perform the notification broadcasting in the terminal of the cluster root. Figure 7.14~7.17 illustrates that each of the cluster children receives the broadcasting notification. Figure 18 presents that the chatting registry server does not have any changes in the procedure. It proves that the cluster is made up with peers in the distributed model of P2P such that the interactions among

them do not need the participation of the chatting registry server. So, I will not show the terminal of the chatting registry server before the multicasting testing is done.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:3497 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3497 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:285856 (285.8 KB)  TX bytes:285856 (285.8 KB)

libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8945
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8945> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.109:8945: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
```

Figure 7.15 The child, 109:8945, receives the broadcasting notification, “Hello, World!”

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:92399 errors:0 dropped:0 overruns:0 frame:0
          TX packets:46159 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:57286094 (57.2 MB)  TX bytes:6891281 (6.8 MB)

libing@freescape:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.111:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.111:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.111:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
```

Figure 7.16 The child, 111:8944, receives the broadcasting notification, “Hello, World!”

```
imac — ssh — 85×40
        nd6 options=1<PERFORMNUD>
        media: autoselect <full-duplex>
        status: inactive
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
        ether 0a:1e:df:df:91:0d
        media: autoselect
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ClusterChild/build/cl
asses

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ClusterChild/build/jar/FreeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.107:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.107:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:24:34 PDT 2017
     [java] <192.168.1.107:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:33:39 PDT 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
```

Figure 7.17 The child, 107:8944, receives the broadcasting notification, “Hello, World!”

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatRegistry — ssh — 80×40
libing@freeweb:~/GreatFreeLabs/Clouds/ChatRegistry$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ChatRegistry/bu
ild

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/cl
asses
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/ja
r

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/Chat
Registry/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/j
ar/FreeWeb.jar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:17 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:19 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:29 CST 2017
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:31 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:40 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:42 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:51 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:53 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:20:06 CST 2017
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] CLUSTER_IP_REQUEST received @Sun Jun 18 07:20:06 CST 2017
```

Figure 7.18 The chatting registry server does not have any updates when the notification broadcasting is performed

4.3 Performing the Notification Unicasting

To perform the notification unicasting, you can keep manipulating in the terminal of the cluster root. First of all, you need to quit from the submenu of typing message. Then, the main menu is displayed again. Select the second option, “2) Unicast a notification”. After that, you enter the submenu once more. Just type the message you

want to unicast, for example, “Are you all right?”.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterRoot — ssh — 85×40
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
0
     [java] Your choice: 1
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Broadcast a notification
     [java]     2) Unicast a notification
     [java]     3) Anycast a notification
     [java]     4) Broadcast a request
     [java]     5) Unicast a request
     [java]     6) Anycast a request
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
2
     [java] Your choice: 2
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 2
     [java] Please type your message:
Are you all right?
     [java] You notification is unicast!
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 7.19 The procedure to perform the notification unicasting on the cluster root

```
imac — ssh — 85×40
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
        ether 0a:1e:df:df:91:0d
        media: autoselect
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ClusterChild/build/cl
asses

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ClusterChild/build/jar/FreeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.107:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.107:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:24:34 PDT 2017
     [java] <192.168.1.107:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:33:39 PDT 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.107:8944: HELLO_WORLD_UNICAST_NOTIFICATION received @Sun Jun 1
8 07:41:16 PDT 2017
     [java] HelloWorldUnicastNotificationThread-notification: Are you all right?
```

Figure 7.20 The child, 107:8944, receives the unicasting notification, “Are you all right?”

Figure 7.19 illustrates the updates in the terminal of the cluster root. Meanwhile, since this is a unicasting, there should be only one cluster child that receives the message, “Are you all right?”.

When performing the unicasting, the unique child that receives the message is selected randomly for the specific testing. You can specify the child if you like. But in this case, the child is randomly chosen. Figure 7.20 shows that the child, 107:8944,

receives the message, “Are you all right?”. Well, all of other children get nothing during the procedure. Figure 7.21~7.23 exhibits the testing results.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:3497 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3497 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:285856 (285.8 KB)  TX bytes:285856 (285.8 KB)

libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.109:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
```

Figure 7.21 The child, 109:8944, receives nothing when the notification unicasting is performed

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:3497 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3497 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:285856 (285.8 KB)  TX bytes:285856 (285.8 KB)

libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8945
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8945> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.109:8945: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
```

Figure 7.22 The child, 109:8945, receives nothing when the notification unicasting is performed

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:92399 errors:0 dropped:0 overruns:0 frame:0
          TX packets:46159 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:57286094 (57.2 MB)  TX bytes:6891281 (6.8 MB)

libing@freescape:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.111:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.111:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.111:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
```

Figure 7.23 The child, 111:8944, receives nothing when the notification unicasting is performed

4.4 Performing the Notification Anycasting

You just follow the prompts of the menu to perform the notification anycasting. Figure 7.24 shows the operations in the terminal of the cluster root. For instance, you type “Good Morning” this time.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterRoot — ssh — 85×40
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
0
     [java] Your choice: 2
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Broadcast a notification
     [java]     2) Unicast a notification
     [java]     3) Anycast a notification
     [java]     4) Broadcast a request
     [java]     5) Unicast a request
     [java]     6) Anycast a request
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
3
     [java] Your choice: 3
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 3
     [java] Please type your message:
Good Morning
     [java] You notification is anycast!
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 7.24 The updates in the terminal of the cluster root when performing the notification anycasting

According to the concept of anycasting, if any of the nodes receives the notification, the anycasting should be terminated. For the testing, the child, 109:8944, and the other one, 111:8944, receive the notification, “Good Morning”. Well, others receive nothing. Figure 7.25~7.28 illustrates the procedure of performing the notification anycasting.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
          collisions:0 txqueuelen:1
          RX bytes:285856 (285.8 KB)  TX bytes:285856 (285.8 KB)

libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.109:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.109:8944: HELLO_WORLD_ANYCAST_NOTIFICATION received @Sun Jun 1
8 07:42:34 CST 2017
     [java] HelloWorldAnycastNotificationThread-notification: Good Morning
```

Figure 7.25 The child, 109:8944, receives the anycasting notification, “Good Morning”

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:3497 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3497 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:285856 (285.8 KB)  TX bytes:285856 (285.8 KB)

libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8945
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8945> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.109:8945: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
```

Figure 7.26 The child, 109:8945, receives nothing when the notification anycasting is performed

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
          collisions:0 txqueuelen:1000
          RX bytes:57286094 (57.2 MB)  TX bytes:6891281 (6.8 MB)

libing@freescape:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.111:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.111:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.111:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.111:8944: HELLO_WORLD_ANYCAST_NOTIFICATION received @Sun Jun 1
8 07:42:34 CST 2017
     [java] HelloWorldAnycastNotificationThread-notification: Good Morning
```

Figure 7.27 The child, 111:8944, receives the anycasting notification, "Good Morning"

```
imac — ssh — 85×40
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
        ether 0a:1e:df:df:91:0d
        media: autoselect
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ClusterChild/build/cl
asses

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ClusterChild/build/jar/FreeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.107:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.107:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:24:34 PDT 2017
     [java] <192.168.1.107:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:33:39 PDT 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.107:8944: HELLO_WORLD_UNICAST_NOTIFICATION received @Sun Jun 1
8 07:41:16 PDT 2017
     [java] HelloWorldUnicastNotificationThread-notification: Are you all right?
```

Figure 7.28 The child, 107:8944, receives nothing when the notification anycasting is performed

4.5 Performing the Request Broadcasting

You can select the fourth option, "4) Broadcast a request", from the main menu, to

test the request broadcasting. Figure 7.29 illustrates the updates in the terminal of the cluster root. For example, you type, “What time is it?”, as the query message. After typing the request, each of the cluster children receives the request message, as shown in Figure 7.30~7.33.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterRoot — ssh — 85×40
     [java]     5) Unicast a request
     [java]     6) Anycast a request
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
4
     [java] Your choice: 4
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 4
     [java] Please type your message:
What time is it?
     [java] HELLO_WORLD_BROADCAST_RESPONSE received @Sun Jun 18 07:52:21 CST 2017
     [java] HELLO_WORLD_BROADCAST_RESPONSE received @Sun Jun 18 07:52:21 CST 2017
     [java] HELLO_WORLD_BROADCAST_RESPONSE received @Sun Jun 18 07:52:21 CST 2017
     [java] HELLO_WORLD_BROADCAST_RESPONSE received @Sun Jun 18 07:52:22 CST 2017
     [java] You request is broadcast!
     [java] You got 4 responses
     [java] 1) response = Broadcast response ... What time is it? id: fd2a862c-5000-4
b6d-8a94-2cf0ae01b86b
     [java] 2) response = Broadcast response ... What time is it? id: fd2a862c-5000-4
b6d-8a94-2cf0ae01b86b
     [java] 3) response = Broadcast response ... What time is it? id: d82f2cb0-2236-4
13b-95b6-68263168cc58
     [java] 4) response = Broadcast response ... What time is it? id: fd2a862c-5000-4
b6d-8a94-2cf0ae01b86b
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 7.29 The updates in the terminal of the cluster root when performing the request broadcasting

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.109:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.109:8944: HELLO_WORLD_ANYCAST_NOTIFICATION received @Sun Jun 1
8 07:42:34 CST 2017
     [java] HelloWorldAnycastNotificationThread-notification: Good Morning
     [java] <192.168.1.109:8944: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:52:22 CST 2017
     [java] Query: What time is it?
```

Figure 7.30 The child, 109:8944, receives the broadcasting request, “What time is it?”

In addition, according to Figure 7.29, different from the tests about notifications, this

time the cluster root receives six responses from each of the cluster children. It proves each cluster child sends their responses to the cluster root, i.e., the request broadcasting is accomplished successfully.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
          collisions:0 txqueuelen:1
          RX bytes:285856 (285.8 KB)  TX bytes:285856 (285.8 KB)

libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8945
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8945> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.109:8945: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.109:8945: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:52:22 CST 2017
     [java] Query: What time is it?
```

Figure 7.31 The child, 109:8945, receives the broadcasting request, “What time is it?”

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
libing@freescape:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.111:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.111:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.111:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.111:8944: HELLO_WORLD_ANYCAST_NOTIFICATION received @Sun Jun 1
8 07:42:34 CST 2017
     [java] HelloWorldAnycastNotificationThread-notification: Good Morning
     [java] <192.168.1.111:8944: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:52:21 CST 2017
     [java] Query: What time is it?
```

Figure 7.32 The child, 111:8944, receives the broadcasting request, “What time is it?”

```
imac — ssh — 85×40
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ClusterChild/build/cl
asses

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ClusterChild/build/jar/FreeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.107:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.107:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:24:34 PDT 2017
     [java] <192.168.1.107:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:33:39 PDT 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.107:8944: HELLO_WORLD_UNICAST_NOTIFICATION received @Sun Jun 1
8 07:41:16 PDT 2017
     [java] HelloWorldUnicastNotificationThread-notification: Are you all right?
     [java] <192.168.1.107:8944: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:56:50 PDT 2017
     [java] Query: What time is it?
```

Figure 7.33 The child, 107:8944, receives the broadcasting request, “What time is it?”

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterRoot — ssh — 85×40
     [java] Input an option:
0
     [java] Your choice: 4
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Broadcast a notification
     [java]     2) Unicast a notification
     [java]     3) Anycast a notification
     [java]     4) Broadcast a request
     [java]     5) Unicast a request
     [java]     6) Anycast a request
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
5
     [java] Your choice: 5
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 5
     [java] Please type your message:
Are you hungry?
     [java] HELLO_WORLD_UNICAST_RESPONSE received @Sun Jun 18 08:06:28 CST 2017
     [java] You request is unicast!
     [java] You got 1 responses
     [java] 1) response = Unicast response ... Are you hungry? id: 49e14fe7-45d5-4f77
-a32a-c4cdad2dd7d7
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 7.34 The cluster root performs the request unicasting and gets one response only

4.6 Performing the Request Unicasting

Let us take the next interesting test. This time we check whether the request unicasting works or not. You must be familiar with the menu. Select the fifth option, “5) Unicast a request”, from the main menu. This time, you can type, “Are you

hungry?", to send the query to the cluster. Figure 3.34 exhibits what happens in the terminal of the cluster root. Since it is the so-called unicasting, only one request is sent to one particular cluster child. In our case, it is selected randomly. According to Figure 3.34, the cluster root receives one response only. That is correct.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.109:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.109:8944: HELLO_WORLD_ANYCAST_NOTIFICATION received @Sun Jun 1
8 07:42:34 CST 2017
     [java] HelloWorldAnycastNotificationThread-notification: Good Morning
     [java] <192.168.1.109:8944: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:52:22 CST 2017
     [java] Query: What time is it?
     [java] <192.168.1.109:8944: HELLO_WORLD_UNICAST_REQUEST received @Sun Jun 18 08:
06:28 CST 2017
     [java] Query: Are you hungry?
```

Figure 7.35 The child, 109:8944, receives the unicasting request, "Are you hungry?"

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
          collisions:0 txqueuelen:1
          RX bytes:285856 (285.8 KB)  TX bytes:285856 (285.8 KB)

libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8945
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8945> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.109:8945: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.109:8945: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:52:22 CST 2017
     [java] Query: What time is it?
```

Figure 7.36 The child, 109:8945, receives nothing when the cluster root performs the request unicasting

Figure 3.35~3.38 demonstrates the updates in the terminals of all of the cluster children. You should find that only one terminal get updated. In this case, the child, 109:8944, is the lucky one. Till now, it proves that the request unicasting runs

successfully!

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
libing@freescape:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.111:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.111:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.111:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.111:8944: HELLO_WORLD_ANYCAST_NOTIFICATION received @Sun Jun 1
8 07:42:34 CST 2017
     [java] HelloWorldAnycastNotificationThread-notification: Good Morning
     [java] <192.168.1.111:8944: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:52:21 CST 2017
     [java] Query: What time is it?
```

Figure 7.37 The child, 111:8944, receives nothing when the cluster root performs the request unicasting

```
imac — ssh — 85×40
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ClusterChild/build/cl
asses

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ClusterChild/build/jar/FreeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.107:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.107:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:24:34 PDT 2017
     [java] <192.168.1.107:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:33:39 PDT 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.107:8944: HELLO_WORLD_UNICAST_NOTIFICATION received @Sun Jun 1
8 07:41:16 PDT 2017
     [java] HelloWorldUnicastNotificationThread-notification: Are you all right?
     [java] <192.168.1.107:8944: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:56:50 PDT 2017
     [java] Query: What time is it?
```

Figure 7.38 The child, 107:8944, receives nothing when the cluster root performs the request unicasting

4.7 Performing the Request Anycasting

The last test we need to take is to verify how the request anycasting works. From the

main menu, you should select the sixth option, “6) Anycast a request”. And then, you can type a query, for example, “What is cloud computing?”. After the query is send out, you get two responses. Because the cluster contains four children, it proves that the anycast is validated successfully. Figure 7.39 illustrates the procedure.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterRoot — ssh — 85×40
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Broadcast a notification
     [java]     2) Unicast a notification
     [java]     3) Anycast a notification
     [java]     4) Broadcast a request
     [java]     5) Unicast a request
     [java]     6) Anycast a request
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
6
     [java] Your choice: 6
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 6
     [java] Please type your message:
What is cloud computing?
     [java] HELLO_WORLD_ANYCAST_RESPONSE received @Sun Jun 18 08:27:26 CST 2017
     [java] HELLO_WORLD_ANYCAST_RESPONSE received @Sun Jun 18 08:27:26 CST 2017
     [java] You request is anycast!
     [java] You got 2 responses
     [java] 1) response = Anycast response ... What is cloud computing? id: 5febbdc8-
8838-4055-b3fe-501bdf05f75f
     [java] 2) response = Anycast response ... What is cloud computing? id: 15570f1f-
f3d0-47a7-96de-359a2ee516d2
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 7.39 The cluster root performs the request anycasting to its cluster and gets two responses

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.109:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.109:8944: HELLO_WORLD_ANYCAST_NOTIFICATION received @Sun Jun 1
8 07:42:34 CST 2017
     [java] HelloWorldAnycastNotificationThread-notification: Good Morning
     [java] <192.168.1.109:8944: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:52:22 CST 2017
     [java] Query: What time is it?
     [java] <192.168.1.109:8944: HELLO_WORLD_UNICAST_REQUEST received @Sun Jun 18 08:
06:28 CST 2017
     [java] Query: Are you hungry?
     [java] <192.168.1.109:8944: HELLO_WORLD_ANYCAST_REQUEST received @Sun Jun 18 08:
27:26 CST 2017
     [java] Query: What is cloud computing?
```

Figure 7.40 The child, 109:8944, receives the anycasting request, “What is cloud computing?”

Among those children, the ones, 109:8944 and 111:8944, get the anycasting request,

"What is cloud computing?" and reply to the cluster root immediately. In contrast, the other two, 109:8945 and 107:8944, receive nothing during the procedure. Figure 7.40~7.43 demonstrates the updates in the four terminals.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
          collisions:0 txqueuelen:1
          RX bytes:285856 (285.8 KB)  TX bytes:285856 (285.8 KB)

libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8945
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8945> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.109:8945: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.109:8945: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:52:22 CST 2017
     [java] Query: What time is it?
```

Figure 7.41 The child, 109:8945, receives nothing during the procedure of the request anycasting

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.111:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.111:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.111:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.111:8944: HELLO_WORLD_ANYCAST_NOTIFICATION received @Sun Jun 1
8 07:42:34 CST 2017
     [java] HelloWorldAnycastNotificationThread-notification: Good Morning
     [java] <192.168.1.111:8944: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:52:21 CST 2017
     [java] Query: What time is it?
     [java] <192.168.1.111:8944: HELLO_WORLD_ANYCAST_REQUEST received @Sun Jun 18 08:
27:26 CST 2017
     [java] Query: What is cloud computing?
```

Figure 7.42 The child, 111:8944, receives the anycasting request, "What is cloud computing?"

```
imac — ssh — 85×40
        status: inactive
greatfree@Mum.local: /GreatFreeLabs/Clouds/ClusterChild$ ant
Buildfile: /GreatFreeLabs/Clouds/ClusterChild/build.xml

clean:
   [delete] Deleting directory /GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ClusterChild/build/cl
asses

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ClusterChild/build/jar/FreeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.107:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.107:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:24:34 PDT 2017
     [java] <192.168.1.107:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:33:39 PDT 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.107:8944: HELLO_WORLD_UNICAST_NOTIFICATION received @Sun Jun 1
8 07:41:16 PDT 2017
     [java] HelloWorldUnicastNotificationThread-notification: Are you all right?
     [java] <192.168.1.107:8944: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:56:50 PDT 2017
     [java] Query: What time is it?
```

Figure 7.43 The child, 107:8944, receives nothing during the procedure of the request anycasting

4.8 Terminating the System

That is great. We are almost done! Now let us terminate the system. If a system cannot be closed properly, I do not think it is a qualified one. If a system cannot be maintained well, it cannot be shut down normally.

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatAdmin — ssh — 85×40

clean:
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ChatAdmin
/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatAdmin/build/jar/FreeW
eb.jar

run:
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Stop CS Chatting Server
     [java]     2) Stop Chatting Registry Server
     [java]     3) Stop Cluster Children
     [java]     0) End
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
3
     [java] Your choice is: 3
     [java]     3) Stop Cluster Children
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Stop CS Chatting Server
     [java]     2) Stop Chatting Registry Server
     [java]     3) Stop Cluster Children
     [java]     0) End
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
```

Figure 7.44 The administrator selects the option, “3) Stop Cluster Children”

4.8.1 Shutting Down the Cluster Children

Do you still remember the administrator? Yes, we can terminate the system by invoking it. Figure 7.44 illustrates we can employ the Ant project, ChatAdmin, to attain the task. After the project is started, there is a menu you can determine which system you want to stop. For our case, we should terminate all of the cluster children at first. So the option three, "3) Stop Cluster Children", is what we need to choose.

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatRegistry — ssh — 80×40
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/cl
asses
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/ja
r

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/Chat
Registry/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/j
ar/FreeWeb.jar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:17 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:19 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:29 CST 2017
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:31 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:40 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:42 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:51 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:53 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:20:06 CST 2017
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] CLUSTER_IP_REQUEST received @Sun Jun 18 07:20:06 CST 2017
     [java] PEER_CHAT_PARTNER_REQUEST received @Sun Jun 18 08:45:00 CST 2017
     [java] UNREGISTER_PEER_REQUEST received @Sun Jun 18 08:45:00 CST 2017
     [java] removed port = 8944
     [java] UNREGISTER_PEER_REQUEST received @Sun Jun 18 08:45:00 CST 2017
     [java] removed port = 8945
     [java] UNREGISTER_PEER_REQUEST received @Sun Jun 18 08:45:00 CST 2017
     [java] removed port = 8944
     [java] UNREGISTER_PEER_REQUEST received @Sun Jun 18 08:45:00 CST 2017
     [java] removed port = 8944
```

Figure 7.45 The chatting registry server updates during the procedure to shut down all of the children within the cluster

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterRoot — ssh — 85×40
     [java]     1) Broadcast a notification
     [java]     2) Unicast a notification
     [java]     3) Anycast a notification
     [java]     4) Broadcast a request
     [java]     5) Unicast a request
     [java]     6) Anycast a request
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
6
     [java] Your choice: 6
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
1
     [java] Your choice: 6
     [java] Please type your message:
What is cloud computing?
     [java] HELLO_WORLD_ANYCAST_RESPONSE received @Sun Jun 18 08:27:26 CST 2017
     [java] HELLO_WORLD_ANYCAST_RESPONSE received @Sun Jun 18 08:27:26 CST 2017
     [java] You request is anycast!
     [java] You got 2 responses
     [java] 1) response = Anycast response ... What is cloud computing? id: 5febbdc8-
8838-4055-b3fe-501bdf05f75f
     [java] 2) response = Anycast response ... What is cloud computing? id: 15570f1f-
f3d0-47a7-96de-359a2ee516d2
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] SHUTDOWN_CHILDREN_ADMIN_NOTIFICATION received @Sun Jun 18 08:45:00 CST 20
17
```

Figure 7.46 The cluster root performs a notification broadcasting to send the termination instruction to all of the children within the cluster

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.109:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.109:8944: HELLO_WORLD_ANYCAST_NOTIFICATION received @Sun Jun 1
8 07:42:34 CST 2017
     [java] HelloWorldAnycastNotificationThread-notification: Good Morning
     [java] <192.168.1.109:8944: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:52:22 CST 2017
     [java] Query: What time is it?
     [java] <192.168.1.109:8944: HELLO_WORLD_UNICAST_REQUEST received @Sun Jun 18 08:
06:28 CST 2017
     [java] Query: Are you hungry?
     [java] <192.168.1.109:8944: HELLO_WORLD_ANYCAST_REQUEST received @Sun Jun 18 08:
27:26 CST 2017
     [java] Query: What is cloud computing?
     [java] <192.168.1.109:8944: SHUTDOWN_CHILDREN_BROADCAST_NOTIFICATION received @S
un Jun 18 08:45:00 CST 2017

main:

BUILD SUCCESSFUL
Total time: 89 minutes 49 seconds
libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$
```

Figure 7.47 The child, 109:8944, is shut down

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40
   [delete] Deleting directory /home/libing/GreatFreeLabs/Clouds/ClusterChild/build

init:
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.109:8945
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.109:8945> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.109:8945: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.109:8945: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:52:22 CST 2017
     [java] Query: What time is it?
     [java] <192.168.1.109:8945: SHUTDOWN_CHILDREN_BROADCAST_NOTIFICATION received @S
un Jun 18 08:45:00 CST 2017

main:

BUILD SUCCESSFUL
Total time: 89 minutes 37 seconds
libing@mother:~/GreatFreeLabs/Clouds/ClusterChild$
```

Figure 7.48 The child, 109:8955, is shut down

After the option is selected, all of the cluster children are shut down. You can see some updates in the terminals of the chatting registry server and the cluster root. Since the cluster children are dominated by the cluster root. The administrator needs to send the instruction to the cluster root. However, how does it know the IP address of the cluster root? So it has to query with the registry server. You can see the registry server receives a request, PEER_CHAT_PARTNER_REQUEST. It is queried by the

administrator. After getting the IP address of the cluster root, the administrator is able to send the instruction to it. After it gets the instruction, the root broadcasts the instruction to each of the cluster children to terminate all of them. Before those children terminate, they need to unregister themselves from the registry server. You can see four requests, UNREGISTER_PEER_REQUEST, are shown in the terminal of the chatting registry server. Figure 7.44~7.50 demonstrates the procedure.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterChild — ssh — 85×40

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/ClusterCh
ild/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ClusterChild/build/jar/Fr
eeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.111:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.111:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:20:06 CST 2017
     [java] <192.168.1.111:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:29:11 CST 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.111:8944: HELLO_WORLD_ANYCAST_NOTIFICATION received @Sun Jun 1
8 07:42:34 CST 2017
     [java] HelloWorldAnycastNotificationThread-notification: Good Morning
     [java] <192.168.1.111:8944: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:52:21 CST 2017
     [java] Query: What time is it?
     [java] <192.168.1.111:8944: HELLO_WORLD_ANYCAST_REQUEST received @Sun Jun 18 08:
27:26 CST 2017
     [java] Query: What is cloud computing?
     [java] <192.168.1.111:8944: SHUTDOWN_CHILDREN_BROADCAST_NOTIFICATION received @S
un Jun 18 08:45:00 CST 2017

main:

BUILD SUCCESSFUL
Total time: 89 minutes 29 seconds
libing@freescape:~/GreatFreeLabs/Clouds/ClusterChild$
```

Figure 7.49 The child, 111:8944, is shut down

```
imac — ssh — 85×40
init:
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/classes
    [mkdir] Created dir: /GreatFreeLabs/Clouds/ClusterChild/build/jar

compile:
    [javac] Compiling 860 source files to /GreatFreeLabs/Clouds/ClusterChild/build/cl
asses

jar:
      [jar] Building jar: /GreatFreeLabs/Clouds/ClusterChild/build/jar/FreeWeb.jar

run:
     [java] Starting up the child ...
     [java]
     [java] ===================
     [java] Child IP Address: 192.168.1.107:8944
     [java] ===================
     [java]
     [java] The child is started ...
     [java] <192.168.1.107:8944> ROOT_IPADDRESS_BROADCAST_NOTIFICATION received @Sun
Jun 18 07:24:34 PDT 2017
     [java] <192.168.1.107:8944: HELLO_WORLD_BROADCAST_NOTIFICATION received @Sun Jun
 18 07:33:39 PDT 2017
     [java] HelloWorldBroadcastNotificationThread-BroadcastNotification received: Hel
lo, World!
     [java] <192.168.1.107:8944: HELLO_WORLD_UNICAST_NOTIFICATION received @Sun Jun 1
8 07:41:16 PDT 2017
     [java] HelloWorldUnicastNotificationThread-notification: Are you all right?
     [java] <192.168.1.107:8944: HELLO_WORLD_BROADCAST_REQUEST received @Sun Jun 18 0
7:56:50 PDT 2017
     [java] Query: What time is it?
     [java] <192.168.1.107:8944: SHUTDOWN_CHILDREN_BROADCAST_NOTIFICATION received @S
un Jun 18 08:49:29 PDT 2017

main:

BUILD SUCCESSFUL
Total time: 89 minutes 19 seconds
greatfree@Mum.local: /GreatFreeLabs/Clouds/ClusterChild$
```

Figure 7.50 The child, 107:8944, is shut down

4.8.2 Shutting Down the Cluster Root

Next, you should shut down the cluster as well. Since the cluster root has the option of quitting, you just choose it to terminate the root. Figure 7.51 demonstrates the procedure. After it is shut down, the registry server gets one more unregistering request, as shown in Figure 7.52.

```
imac — libing@freescape: ~/GreatFreeLabs/Clouds/ClusterRoot — ssh — 85×40
     [java] Please type your message:
What is cloud computing?
     [java] HELLO_WORLD_ANYCAST_RESPONSE received @Sun Jun 18 08:27:26 CST 2017
     [java] HELLO_WORLD_ANYCAST_RESPONSE received @Sun Jun 18 08:27:26 CST 2017
     [java] You request is anycast!
     [java] You got 2 responses
     [java] 1) response = Anycast response ... What is cloud computing? id: 5febbdc8-
8838-4055-b3fe-501bdf05f75f
     [java] 2) response = Anycast response ... What is cloud computing? id: 15570f1f-
f3d0-47a7-96de-359a2ee516d2
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Type Message To:
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
     [java] SHUTDOWN_CHILDREN_ADMIN_NOTIFICATION received @Sun Jun 18 08:45:00 CST 20
17
0
     [java] Your choice: 6
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Broadcast a notification
     [java]     2) Unicast a notification
     [java]     3) Anycast a notification
     [java]     4) Broadcast a request
     [java]     5) Unicast a request
     [java]     6) Anycast a request
     [java]     0) Quit
     [java] ========== Menu Tail ===========
     [java]
0
     [java] Your choice: 0

main:

BUILD SUCCESSFUL
Total time: 113 minutes 17 seconds
libing@freescape:~/GreatFreeLabs/Clouds/ClusterRoot$
```

Figure 7.51 The cluster root is shut down

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatRegistry — ssh — 80×40
    [mkdir] Created dir: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/ja
r

compile:
    [javac] Compiling 860 source files to /home/libing/GreatFreeLabs/Clouds/Chat
Registry/build/classes

jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/j
ar/FreeWeb.jar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:17 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:19 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:29 CST 2017
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:31 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:40 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:42 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:51 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:53 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:20:06 CST 2017
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] CLUSTER_IP_REQUEST received @Sun Jun 18 07:20:06 CST 2017
     [java] PEER_CHAT_PARTNER_REQUEST received @Sun Jun 18 08:45:00 CST 2017
     [java] UNREGISTER_PEER_REQUEST received @Sun Jun 18 08:45:00 CST 2017
     [java] removed port = 8944
     [java] UNREGISTER_PEER_REQUEST received @Sun Jun 18 08:45:00 CST 2017
     [java] removed port = 8945
     [java] UNREGISTER_PEER_REQUEST received @Sun Jun 18 08:45:00 CST 2017
     [java] removed port = 8944
     [java] UNREGISTER_PEER_REQUEST received @Sun Jun 18 08:45:00 CST 2017
     [java] removed port = 8944
     [java] UNREGISTER_PEER_REQUEST received @Sun Jun 18 09:13:14 CST 2017
     [java] removed port = 8945
```

Figure 7.52 The cluster root shuts down and the registry server receives the unregistering request

4.8.3 Shutting Down the Registry Server

The chatting registry server is started first and shut down later than all of other components in the system. You can see its important position for a distributed system.

To shut it down, you need to select the second option, “2) Stop Chatting Registry Server”, from the menu of the administrator. After that, the registry server is terminated as shown in Figure 7.53.

```
imac — libing@freeweb: ~/GreatFreeLabs/Clouds/ChatRegistry — ssh — 80×40
jar:
      [jar] Building jar: /home/libing/GreatFreeLabs/Clouds/ChatRegistry/build/j
ar/FreeWeb.jar

run:
     [java] Chatting registry server starting up ...
     [java] Chatting registry server started ...
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:17 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:19 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:29 CST 2017
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:31 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:40 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:42 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:15:51 CST 2017
     [java] PEER_CHAT_REGISTRY_REQUEST received @Sun Jun 18 07:15:53 CST 2017
     [java] REGISTER_PEER_REQUEST received @Sun Jun 18 07:20:06 CST 2017
     [java] 2) getIdlePort(): port size = 1
     [java] 2) getIdlePort(): port = 8944
     [java] CLUSTER_IP_REQUEST received @Sun Jun 18 07:20:06 CST 2017
     [java] PEER_CHAT_PARTNER_REQUEST received @Sun Jun 18 08:45:00 CST 2017
     [java] UNREGISTER_PEER_REQUEST received @Sun Jun 18 08:45:00 CST 2017
     [java] removed port = 8944
     [java] UNREGISTER_PEER_REQUEST received @Sun Jun 18 08:45:00 CST 2017
     [java] removed port = 8945
     [java] UNREGISTER_PEER_REQUEST received @Sun Jun 18 08:45:00 CST 2017
     [java] removed port = 8944
     [java] UNREGISTER_PEER_REQUEST received @Sun Jun 18 08:45:00 CST 2017
     [java] removed port = 8944
     [java] UNREGISTER_PEER_REQUEST received @Sun Jun 18 09:13:14 CST 2017
     [java] removed port = 8945
     [java] SHUTDOWN_CHAT_SERVER_NOTIFICATION received @Sun Jun 18 09:20:22 CST
2017

main:

BUILD SUCCESSFUL
Total time: 126 minutes 18 seconds
libing@freeweb:~/GreatFreeLabs/Clouds/ChatRegistry$
```

Figure 7.53 The chatting registry server is shut down

```
imac — libing@mother: ~/GreatFreeLabs/Clouds/ChatAdmin — ssh — 85×40
     [java] ========== Menu Head ===========
     [java]     1) Stop CS Chatting Server
     [java]     2) Stop Chatting Registry Server
     [java]     3) Stop Cluster Children
     [java]     0) End
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
3
     [java] Your choice is: 3
     [java]     3) Stop Cluster Children
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Stop CS Chatting Server
     [java]     2) Stop Chatting Registry Server
     [java]     3) Stop Cluster Children
     [java]     0) End
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
2
     [java] Your choice is: 2
     [java]     2) Stop Chatting Registry Server
     [java]
     [java] ========== Menu Head ===========
     [java]     1) Stop CS Chatting Server
     [java]     2) Stop Chatting Registry Server
     [java]     3) Stop Cluster Children
     [java]     0) End
     [java] ========== Menu Tail ===========
     [java]
     [java] Input an option:
0
     [java] Your choice is: 0

main:

BUILD SUCCESSFUL
Total time: 39 minutes 45 seconds
libing@mother:~/GreatFreeLabs/Clouds/ChatAdmin$
```

Figure 7.54 The administrator is shut down

4.8.4 Shutting Down the Administrator

The last step is to shut down the administrator. It is simple since it has the option of "0) End". Select it and then the entire system is terminated, as shown in Figure 7.54.

5. A Summary

I believe the technique of cluster programming is always a challenge topic. Fortunately, GreatFree provides you with sufficient support in terms of APIs and patterns such that the entire procedure becomes straightforward and flexible. Although the system programmed in the chapter is a contrived one, you learn all of the new APIs and new patterns for cluster programming in GreatFree. Based on the one-to-many interaction distributed model, you can come up with more complicated distributed systems in terms of the scale and the diversity of applications through simplified behaviors.

References

[1] Chuck Lam. Hadoop In Action. Manning Publications Co., 2011, ISBN: 978-1-935-18219-1.

[2] Solr. 2015. DOI: http://lucene.apache.org/solr/.

[3] Tomcat. 2015. Apache Tomcat. DOI: http://tomcat.apache.org/.

[4] HTTP. 1996. HTTP – Hypertext Transfer Protocol Overview. DOI: .http://www.w3.org/Protocols/.