NetComputing :: The Social Hosting Project

Samuel Esposito and Jorn van de Beek

2010-04-01

1 Context

The social hosting project is a project that means to revolutionize the traditional client-server pattern that is so widely used for web-serving on the internet today. In this pattern typically different clients connect to one server and this server handles all the clients requests itself. The software developed for this project drastically changes this model by letting the clients help the server out in hosting.

In the next sections we discuss the problems that arise in the traditional client-server approach, how our software can solve them, the design of our solution, our implementation in the Java programming language and the benefits and shortcomings of our work.

2 Problem statement

The problem with the traditional client-server pattern as we described it above is that one server has to handle the load of all clients requests itself. Since it is impossible to forecast the exact amount of requests that one server will receive from all clients on the internet, it is impossible wo equip a server with the right amount of resources and it often happens that a server is unable to fullfill the clients requests within an appropriate amount of time (cfr. the frontpage of popular websites), resulting in a denail of service by this server.
In our approach this problem is solved by letting the clients help the server out in hosting. The larger the group of clients that connect to the server and make use of its services, the bigger will be the group of social hosts (the clients that take initiative to help the server out in hosting). So when a server is under the load of many request, it can simply delegate part of this load over the social hosts without needing any extra resources and without denying services to clients.

3 Solution design

As we said before the social hosting project means to replace the traditional client-server pattern in which there is a server running on one machine (hereafter referred to as the server-side) listening for HTTP-requests and one or more clients on other machines on the same network (referred to as the client-side hereafter) doing HTTP-requests on this server. The architecture of the software build for this project consists of different components, two of them running on the server-side and two of them on the client-side.

3.1 Server-Side HTTP-Server

On the server-side there will be a component listening for incoming HTTP-requests, just like in the traditional client-server pattern. This server-side HTTP-server however will not handle all HTTP-requests by itself. If it disposes over a pool of helper HTTP-servers running on other hosts on the network, it will propagate the request to the servers on these social hosts, thus balancing the load. The term we coined for this process is 'social hosting', because the pool of hosts is dynamic and consists of social hosts that offer hosting services on their own initiative to the sever-side HTTP-server. To manage these offered services, we need another component, which is discussed in the next subsection.

3.2 Sever-Side Registration-Server

As discussed above we need a component that registers and manages all incoming offers of hosting services. This process is handled by a server-side registration server that is continuously listening for offers from social hosts that want to share the load of HTTP-requests with the server-side HTTP-server. When a request is received, it is stored in a data structure that is available to the HTTP-server running on the server-side. This HTTP-server then propagates part of its HTTP-requests to the registered social host. Now the question arises, what are the components running on the social host? This is lined out in the next two subsections.

3.3 Client-Side Registration-Client

The social hosts we introduced earlier are actually the clients from the traditional client-server pattern. These clients can take the initiative to help the HTTP-server on the server-side out in handling HTTP-requests by registering themselves as social hosts. This registration is done by the registration client that makes a request to the registration server on the server-side, passing all information necessary for the future propagation of HTTP-requests.

3.4 Client-Side HTTP-Server

This leaves us only the handling of the propagated HTTP-requests by the social host. It comes as no surprise that this will be done by an HTTP-server on the client-side. This server disposes over (part of) the same resources the server-side HTTP-Server uses to answer the HTTP-requests. Also it knows the origin of the requests, so it can send the answer directly to the client that made the request. This client in turn can make future requests directly to the social hosts until the social host again propagates the request to the server-side HTTP-server and we're back to square one.

In the next section the implementation of the different components is discussed.

4 Overview of realization

In the previous section we explained of what components the social hosting software consists and how they interact. In this section we will elaborate each components implementation in the Java programming language. The Java code displayed in the listings contain no working code but rather a summary of the subsequent commands that implement a certain process.

To ease the communication and cooperation between the two components running on the server-side, we choose to implement them as different threads in the same java process, sharing a portion of memory for communication.

```
1 // Shared memory for social hosts
2 private static Vector<String> socialHostIPs = new Vector<String>();
3 HTTPServer httpServer = new HTTPServer(port);
4 httpServer.start(); // Start HTTP-server
5 RMIServer rmiServer = new RMIServer();
6 rmiServer.run(); // Start RMI-server
```

4.1 Server-Side HTTP-Server

The server-side HTTP-server is multi-threaded. One Java thread opens a socket on a port. For every request it receives on this socket, it opens a new thread and passes the request to it.

```
1 serverSocket = new ServerSocket(port);
2 while (listening)
3   new HTTPServerThread(serverSocket.accept()).start();
```

The thread handling the request reads the requests header from the socket into and HTTPRequest object implementing the HTTP 1.1 specification for a request. Then it tries to retreive a social host from shared memory. If it receives a host, it compiles an HTTPResponse object implementing an HTTP redirect.

If there is no social host available, it will handle the request itself conform the HTTP 1.1 response specification.

```
HTTPRequest request = readSocket(); // read request
HTTPResponse response = new HTTPResponse();
String socialHost = Server.randomHost(); // acquire social host
if (socialHost != null) { // a social host is available
  response.setResponseCode(HTTPResponse.REDIRECT); // redirect
  response.setContentType(HTTPResponse.HTML_MIME);
  response.setContentLength(Integer.toString(content.length));
  response.setLocation("http://" + socialHost + request.path());
} else { // no social host available
  FileHandler file = new FileHandler(request.path()); // fetch file
  response.setResponseCode(file.serverStatus());
  response.setContentType(file.mimeType());
  response.setContentLength(Integer.toString(content.length));
}
```

4.2 Server-Side RMI-Server

The registration of social hosts is implemented using Java Remote Method Invocation (RMI). The server-side RMI-server runs in a Java thread. First it sets the security manager monitoring the executions that are triggered by a remote invocation and matching them with the security policy provided. Then it binds the RMI-server to the RMI Registry so that clients know where to access its services.

```
System.setSecurityManager(new RMISecurityManager());
Naming.rebind(RMIServerName, this);
```

The RMI-server provides two methods, one for registering a social host, and one for deregistering it. When a host is registered, the location of its hosting service is added to a list in the memory the RMI-server shares with the server-side HTTP-server. The HTTP-server can now at any time query for this social host and propagate its HTTP-requests to it through an HTTP redirect. When a social host is diregistered, it is removed from the list in shared memory and cannot be queried any more by the HTTP-server.

```
// Register a social host
public boolean registerSocialHost(int port) throws RemoteException {
  Server.addSocialHost(host);
}
// Deregister social host
public boolean deregisterSocialHost(int port) throws RemoteException {
  Server.removeSocialHost(host);
}
```

The components on the client-side are implemented as threads in a Java Applet, hosted by the HTTP-server on the server-side and running as a separate process initiated by the browser on the client-side. All data (html-files and the like) necessary for handling the HTTP-requests that are propagated from the server-side HTTP-server is contained in the Applets jarfile. The components share the applets interface for the invocation of their methods and for the logging of their actions. In order to give the applet enough privileges to run an HTTP-server itself, it was neccessary to sign it. The applet has a simple interface with a start and stop button, a text-area to log the performed actions and the HTTP-requests and responses, a clear log button and a text-field to optionally choose a port to run the HTTP-server on.

```
startButton = new Button("Start Server"); // Start button
stopButton = new Button("Stop Server"); // Stop button
// Text area for logging
textArea = new TextArea("", 10, 73, TextArea.SCROLLBARS_VERTICAL_ONLY);
clearButton = new Button("Clear Log"); // Clear button
textField = new TextField(5); // Text field for port number
```

When the start button is pressed, the HTTP-server is fired up.

```
1 public void actionPerformed(ActionEvent evt) { // Event handler
2   if (evt.getSource() == startButton) { // The start button is pressed
3     port = textField.getText();
4     startButton.setEnabled(false);
5     stopButton.setEnabled(false);
6     server = new HTTPServer(this, port);
7     server.start(); // The HTTP-server is started
8   }
9 }
```

4.3 Client-Side HTTP-Server

The HTTP-server on the client-side is very similar to the one on the server-side, except in that is doesn't propagate its HTTP-requests. One thread opens a socket on a specified port. When this is successfull, it commands the RMI-client to register the host. Then for every HTTP-request it gets propagated from the server-side HTTP-server, a new thread is fired up handling the request.

```
1 RMIRegistrar registrar = new RMIRegistrar(log);
2 serverSocket = new ServerSocket(port); // Open socket for HTTP-server
3 boolean success = registrar.registerSocialHost(port); // Register social host
4 while (listening)
5   new HTTPServerThread(applet, serverSocket.accept()).start();
```

The thread handling the request then reads the HTTP-request from the socket, loads the requested files from the Applets jar, compiles an HTTP-reponse conform the HTTP 1.1 specification and writes it to the socket.

```
1 HTTPRequest request = readSocket(); // Read request
2 InputStream is = applet.getClass().getResourceAsStream(filePath); // Get file from jar
3 is.read(content, offset, bufferSize); // Load file
4 // Compile HTTP-response
5 HTTPResponse response = new HTTPResponse(file.serverStatus(),
6                                           file.mimeType(),
7                                           Integer.toString(content.length));
8 OutputStream out = socket.getOutputStream();
9 out.write(response.header().getBytes()); // Write response
10 out.write(content);
```

When the user clicks the 'Stop Server' button on the Applet, the social host is deregistered at the RMI-server and the server socket is closed.

```
1 boolean success = registrar.deregisterSocialHost(port); // Deregister social host
2 serverSocket.close(); // Close server socket
```

4.4 Client-Side RMI-Client

For making a request on the RMI-server, the RMI-client starts by looking up the RMI-server in the RMI Registry on the server-side. It receives back a RMIRemoteRegistration object representing the remote object that implements the registration methods we want to call.

```
1 RMIRemoteRegistration register = (RMIRemoteRegistration) Naming.lookup("//" +
2                                   applet.getCodeBase().getHost() +
3                                   "/SocialHostingRegistry");
```

Registering and deregistering a social host now can simply be done by invoking a method on this RMIRemoteRegistration object and passing it the right arguments.

```
1 // Register social host
2 boolean success = register.registerSocialHost(port);
3 // Deregister social host
4 boolean success = register.deregisterSocialHost(port);
```

5 Evaluation

The software created in the social hosting project could revolutionize the way popular websites are being hosted on the internet by letting frequent visitors help out in handling the huge HTTP traffic these sites get. In this approach popular sites getting many visitors also have lots of potential social hosts and eventually the amount of traffic they get from visitors and the support they receive from social hosts might equal out. No more sites going down because of popularity and no high costs to handle the huge traffic.

Of course our software has some limitations and a lot of aspects are to be refined. One limitation is that the software currently only works for static html pages (with no content retreived from a database). It would require further research to also include distributed databases in the setup.

Another point is the registration of social hosts. Currently a social host is registered when its HTTP-server is started up and deregistered when the server is shut down. The server being shut down because the user closes the browser window is not modeled, resulting in the server-side HTTP-server propagating requests to a server that doesn't exist any more. This problem could be solved by giving the social host a lease for a certain period of time upon registration and requiring to reregister when the lease expires. This approach would also solve the problem of a social host serving outdated pages. Such a host could simply be notified to refresh its server content when it tries to reregister.

An issue closely related to the registration process is the amount of social hosts that is handling requests at the same time. Currently the software propagates all HTTP-requests as soon as it has at least one social host. This is handy for demonstration purposes, but will clearly not work in a real life situation. It could however easily be changed to a system that at all times tries to maintain a certain amount of social hosts (probably related to an estimate of the expected traffic). This can be achieved by always having a portion of the traffic, proportional to the odds that the required amount of social hosts will register, being handled by the server-side HTTP-server itself (note that only the server-side HTTP-server includes the social host Applet in its HTTP-responses).