NetComputing :: The SocialHosting Project

Samuel Esposito and Jorn van de Beek

2010-04-01

1 Context

2 Problem statement

3 Solution design

The social hosting project means to revolutionize the traditional client-server pattern where 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 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 response specification.

```
1 HTTPRequest request = readSocket(); // read request
2 HTTPResponse response = new HTTPResponse();
3 String socialHost = Server.randomHost(); // acquire social host
4 if (socialHost != null) { // a social host is available
5   response.setResponseCode(HTTPResponse.REDIRECT); // redirect
6   response.setContentType(HTTPResponse.HTML_MIME);
7   response.setContentLength(Integer.toString(content.length));
8   response.setLocation("http://" + socialHost + request.path());
9 } else { // no social host available
10   FileHandler file = new FileHandler(request.path()); // fetch file
11   response.setResponseCode(file.serverStatus());
12   response.setContentType(file.mimeType());
13   response.setContentLength(Integer.toString(content.length));
14 }
```

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.

```
1 System.setSecurityManager(new RMISecurityManager());
2 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.

```
1 // Register a social host
2 public boolean registerSocialHost(int port) throws RemoteException {
3   Server.addSocialHost(host);
4 }
5 // Deregister social host
6 public boolean deregisterSocialHost(int port) throws RemoteException {
7   Server.removeSocialHost(host);
8 }
```

The components on the client-side are implemented as threads in a Java Applet running as a separate process initiated by the browser. The components share the applet interface for the invocation of their methods and for the logging of their actions. In order to give the applet enough privileges for running an HTTP-server, it was neccessary to sing it. The applet has a

4.3 Client-Side RMI-Client

4.4 Client-Side HTTP-Server

5 Evaluation