Clock synchronization

Dario Brazzi, Francesco Pegoraro

March 04, 2013

Abstract

Implement in Java a clock synchronization primitive. Choose the strategy that you think best to synchronize the clock of a requesting machine against that of a reference (server) machine.

Contents

Part I
Problem description

1 Problem analysis

The goal of this project is to create a Java server providing a clock synchronization primitive.

The server must implement at least one standard protocol for clock synchronization, possibly more than one. A suitable choice is the Cristian's algorithm. The server will reply to each query with the current time, provided by the OS.

The client must be able to ask to a server the current time and to receive it to be displayed it to the user.

2 Assumptions

- For the Christian's algorithm, the network messages have comparable travel time, possibly identical.

Part II
Solution description

3 Cristian's algorithm

The simplest method to achieve clock synchronization is Cristian's algorithm. It requires a server who knows the correct time, a client that wants to synchronize its clock and a link between the client and the server.

The message exchange is the following:

- The client send a message to the server, requesting the current time, and starts a timer. M_1= (REQ_TIME)
- The server receives message $M_1$and immediately starts a timer. Than, process the request, building the response which is constituted of the current time and the time interval needed to build the response. So, just before sending the response stops the timer, than put the timer value and the current time in the response and send the response back to the client. M_2= (current_time, server_processing_time)
- The client receive message $M_2$from the server and stops its timer. Than, sets its current time to
  t = server_current_time + $\frac{1}{2}$(client_processing_time - server_processing_time)

4 Program Design

4.1 Server

The server must open a ServerSocket and listen to the first free port in the range [4444;4454]. If no free port in this range is available, the server will shut down displaying an error.

After the server has registered, will wait any incoming message from any client, until it's shutdown.

When a message from a client is received, the response is delegated to a new thread of the server, such that is possible to reply to a new client before finishing previous requests. The new thread will receive the connection with the client and the timer started by the parent process, timer that will measure the time took by the server to serve the response. It will reply to the client with the most recent current time and the server processing time, eventually closing the socket with the client.

The server will expose the following methods:

- *setPort()*: sets the server listening port to a specified value
- *getPort()*: returns the current listening port, or the default one if not specified
- *startServer()*: starts the server and opens a ServerSocket on the specified port. If the port is not specified, use one in the range [4444;4454].
- *stopServer()*: if the server is active, stop it and close the open socket.

4.2 Client

The client must be able to connect to a specified server on a specified port, sending a message requesting the current time and receiving it. The time will be presented to the user in the following format: "dd/MM/yyyy hh:mm:ss SSS", for example "15/03/2013 15:06:22 598". The client can also get the current time as milliseconds passed since 01/01/1970, also called "unix time", which is useful in many situations.

The client will expose the following methods:

- *getCurrentTime()*: returns the current time as Unix Time (milliseconds since 01/01/1970)
- *getCurrentTimeAsString()*: returns the current time in the format "dd/MM/yyyy hh:mm:ss SSS"
- *setPort()*: sets the port to which the client will connect to the server
- *getPort()*: returns the current clock server port
- *setServer()*: sets the name or the IP address of the clock server
- *getServer()*: returns the current name or IP address of the clock server

4.3 Comunication Protocol

The protocol defines the role of each agent, in this case the client and the server, and specifies the rules to excange messages between the client and the server.

4.3.1 Server protocol

The server reply to well formatted requests with the same response with the following format:

<current_time_server><separator><elapsed_time_server>

Where

- <current_time_server> is a string containing the current time on the server, in milliseconds since Unix epoch (1 Jan 1970) (see `http://en.wikipedia.org/wiki/Unix_time`)
- <separator> is a string (or a char) separating the two values. It must not be a number. For example, it can be the string ":"
- <elapsed_time_server> is a string containing the time elapsed on the server while the response was being served, in nanoseconds.

An example of a possible response is the following:

1363964626096:8347760

A request is well formatted if the received message is equal to one of the two messages described in the Client Protocol section, one for Simple request and one for Full request.

If a request is not well formatted the message

ERROR! Request not valid.

is sent back to the client,

4.3.2 Client protocol

The client has two types of requests: *Simple* and *Full*.

The Simple request A simple request is composed of just a request followed by the server response. The request is a single message, as defined in the following line

REQ SIMPLE SYNC

The client must start a timer t before sending the message to the server, stopping t as soon as the response is received, storing the time elapsed in the variable *timeElapsedClient*. The timer should have a sensibility of nanoseconds, like the server's.

The client should obtain a message from the server in the format

<current_time_server><separator><elapsed_time_server>

The real current client time is calculated with the formula $time = currentTimeServer + messageTravelTime$, where $messageTravelTime = (timeElapsedClient - timeElapsedServer)/2000$

- *time* is the real current time on the client, espressed in milliseconds since Unix epoch
- *currentTimeServer* is <current_time_server>, in milliseconds since Unix epoch
- *messageTravelTime* is the time spent by a message on the network to go from the client to the server, or vice versa.
- *timeElapsedClient* and *timeElapsedServer* are in nanoseconds. Their difference is equal to $2 * messageTravelTime$, the total roundtrip time. Divided by 2000 it is equal to the time spent by one message to travel from the client to the server, in milliseconds. The fundamental assumption is that the network travel times are comparable, e.g. there are no important disequalities.

The Full request A Full request is simply the repetition of a Simple request N times, with $1 < N < 50$, where the *currentTimeServer* used is the one in the last message and *messageTravelTime* is the mean of all the $messageTravelTime_i$ present in the requests, $i \in [1, N]$.

In a formula, given $i \in [1, N]$ is the response index, $time = currentTimeServer_N + \frac{\sum_{i=1}^{N} messageTravelTime_i}{N}$, where $messageTravelTime_i = (timeElapsedClient_i - timeElapsedServer_i)/2000$.

Between two requests the client should wait at least 100 milliseconds to do not falsate the measure.

5 Conclusions

The program has been succesfully implemented in Java.

SyncServer is the server, where it can be specified the listening port, where it will listen until it's terminated.

SyncClient is the client. It requires the server IP (or logical name) and its port, then it will request the current time to the server and display to the user.

Both are console applications and can be invoked with the command *java -jar [JarName]*.