A CLIENT-SERVER APPLICATION FOR STUDENT TIME-TABLE AUTOMATION

Dr. Marius Dorian ZAHARIA

Associate professor, Computer Science and Engineering Department
University POLITEHNICA of Bucharest
E-mail: zaharia@cs.pub.ro

Abstract: The paper describes the features of a distributed application that assists the process of building time-tables for the students of the Faculty of Control Systems and Computer Science at University POLITEHNICA of Bucharest. Some concrete internal implementation details, design decisions as well as aspects concerning the proper usage of the application are also described.
Keywords: distributed application, client-server application, mutual exclusion, synchronization

1. Introduction

The problem of automating the time table building process knew different solving approaches, however, none of them allowed a complete automation of this above mentioned activity.
The time-table draw-up problem could be compared with the problem of allocation of resources to processes, in a multitasking operating system (see Silberschatz (2002)). The resources to be allocated are the classrooms and the available study time in each classroom. The solution should satisfy requirements that measure the quality of the allocation of time/classrooms to different groups of study. The allocation algorithm must avoid fragmentation in the programs of students and professors as well. In practice, in the present application, the allocation of resources must also satisfy subjective requirements. One process, competing for resources, corresponds to the activity of building the time-table of one formation of study. In the Faculty of Control Systems and Computer Science every such activity is done by a teaching assistant that is specifically assigned to build the time-table of one group of study. He has to:

(1) take into consideration the particular requirements of the teaching professors and to allocate classrooms and time for the educational activities so that (ideally)
(2) no fragmentation occurs in the student time-tables and
(3) allocation of the same classroom for two overlapping time intervals at two different groups of study is avoided.

The requirement (1) is difficult to model in order to achieve the complete automation of the time-table building process. That is why the present application gives to the "time-table agents" only the possibility to allocate/deallocate chunks of the time/classroom resource, keeping track of the current status of this resource. The application is formed from a server that manages the time/classroom resource and answers the requests (formulated by the teaching assistants) of client programs implemented as Java applets.

2. Implementation issues

The common resource is represented by a 3 dimensional array whose dimensions correspond to:

- classrooms,
- time during one day (the index set associated to this dimension is {8, 9, …, 21}, every constant designates the starting moment of a one hour length time interval)
- working days (this third dimension is indexed on the set: {Monday, Tuesday, Wednesday, Thursday, Friday}).

The time dimension of the classroom/time data space was therefore split and represented as two "array dimensions" that codify workdays and respectively hours during one day. Figure 1 shows a two dimensional section in this array that corresponds to one of the days of the week. Every cell of the array denotes a time interval of one hour length associated to a certain classroom. The cell value specifies if the classroom is free or allocated and in this last case what is the ID of the year of study that has allocated (occupied) the classroom. The data modeling the common resource are stored in an ASCII file containing one four-byte record for every array cell. This allows a reduced communication overhead and an easy debug of the

application. The time allocation can be done separately taking into account the even or odd weeks.
Every group of study and consequently every ordinary user receives separate quotas that denote the maximum numbers of hours (for lecture halls and respectively applications classrooms) that the user may allocate.

The main modules of the application fulfill the following functionalities:

User management: That module identifies the users after their names. The set of users is fixed, there is no need to dynamically add or remove a user. The user names are a-priori fixed and, in most cases, coincide to the names of the student groups of study. Every user has an associated password that is kept encrypted in a special data file.

Verification of access rights. This module verifies the rights associated to each user. From this point of view the users are forming three classes:

- The privileged user (called master) that has all the possible rights. It can change the password of any other user, can read and write any information in the database (in fact he can supersede any other user) can modify the maximal quotas of hours that is associated to every user.
- The ordinary users can read (view) the content of the common database, allocate time/classrooms in the limit of the quotas they received from the master user. One ordinary user can also de-allocate time/classrooms but only if these have been previously allocated by itself.
- The user guest (that usually has no password) can only view the content of the database i.e. the diagrams showing how every classroom is occupied at any moment by a certain group of study.

The *resource allocation* module uses the FCFS (First Come First Served) scheduling policy. This kind of discipline is not always proper, especially in the case of lecture halls allocation where probably a priority based allocation would be better suited. In order to assure a fair distribution of time/classrooms between different categories of users (i.e. the control systems and the computer science departments of the Faculty) a pre-allocation of the course classes is initially done.

The *communications management* module. Communications are developed using stream sockets. The application uses programming techniques characteristic to Java environment and based on classes modeling data streams (DataInputStream, DataOutputStream, BufferedReader, BufferedWriter, FileReader, FileWriter). The TCP socket connections are created using the Socket class.
The application has two work-modes: the local mode (when both parts of the application run on the same machine) and the remote mode. The local work-mode is particularly useful when debugging the application.
The server is *multi-threaded* therefore capable of serving more clients concurrently. The server creates a new thread for every new user that connects in the system. The sequence below describes the actions the server performs when a client tries to connect and the way the server interprets the orders received from the client (see Mahmoud (1999)).

```
class Connects extends Thread {
// the class that implements the server is derived
// from class Thread
Socket client;
BufferedReader is;
DataOutputStream os;
GrilaSali grs=new GrilaSali();
static Integer lockObj=new Integer(1);
int qotaCurs[]=new int[Passwd.NUSER];
//lecture hours quotas for every year of study
int qotaSem[]=new int[Passwd.NUSER];
//application hours quotas for every year of study
private boolean flagDbg, flagMono;
//Debug and mono-user are alternative modes of the
//application. Normal mode is Remote.

public Connects(Socket s, int flg)
// the actions performed by the servers
//when a client connects
//are defined in the constructor Connects
{client=s;
  flagMono=false; flagDbg=false;
//the application has three work modes
  switch(flg) {
    case 1: flagMono=true; break;
    case 2: flagDbg=true; break;
    case 3: flagDbg=true; flagMono=true; break;
  }
  if(flagMono) System.out.println("Server started in
monouser mode");
  if(flagDbg) System.out.println("Server started in
debug mode");
  try {
    client.setTcpNoDelay(false);
    is=newBufferedReader(new InputStreamReader
                    (client.getInputStream()));
    os=new
        DataOutputStream(client.getOutputStream());
  } catch(IOException e) {
     try{
       client.close();
```

```
    } catch (IOException ex) {
      System.out.println("Connects -- error "+ex);
    }
   return;
  }
//the access to the common resource is done in
//mutual exclusion
  synchronized(lockObj) {
    if(!grs.ReadSali()) { System.out.println("Grid
File not found"); }
//Grid file is the internal name of the common
//resource
    else { this.start();} // starts the current thread
  } // synchronized
}
```

The method run() interprets the commands received from the clients and sends them back the required results. From the implementation sample below, one can notice the internal format of every command recognized by the server. The implementation of run() method is detailed in case of the Add command (with code 2) that performs the allocation of a classroom/time chunk. The synchronized access to the common database makes possible to achieve a transaction behavior in the client-server dialog.

```
public void run()
//the run method of the Thread class is redefined
//by current thread
{int uid, nrsala, hinit, hfin, flg;
  int cmd[]=new int[10];
  byte bcmd[]=new byte[10];
  while(true) {
     // read command code and parameters
     cmd=readCmd(is);
      synchronized (lockObj) { //mutual exclusion
      switch(cmd[0]) {  //interpret command code

case 0:…
// verify password: 0+uid+encripted_password(8)
case 1:…
// set password: 1+uid+encrypted_password
       case 2:  //Add:
//2+uid+reqfor+no_cls+no_day+hour_init+hour_fin
        grs.ReadSali();
        if(doTranzactie(grs, cmd[1], cmd[1],
                        (char)cmd[2], cmd[3],
                        cmd[4], cmd[5], cmd[6])) {
         grs.WriteSali();
         try {
           os.write(1);
           os.flush();
         } catch (IOException e1) {
         System.out.println("Connects.run() -- answer
write error1");
        }}
        else {
            try {
              os.write(0);
              os.flush();
            } catch (IOException e1) {
         System.out.println("Connects.run() -- answer
write error2");
            }}
        break;
       case 3: … // Remove:
//3+uid+no_cls+no_day+hour_init+hour_fin
       case 4: …// Send the grid contents to the client
       case 5:  //client application ended
         try {client.close();} catch (Exception e){};
         return;
       case 6: … //SetQuota
       case 7: … //GetQuota
       default: System.out.println("Connects run() –
Invalid command!");
     } //switch
    } // synchronized
  } //while(true)
}
```

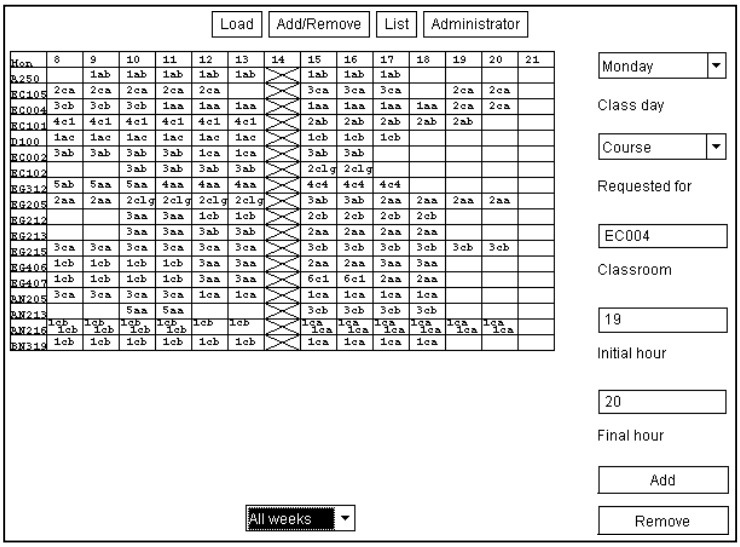

Figure 1. The form used to specify the Add/Remove transaction parameters in the client application

The access to the application database (i.e. to the common resource) is done in mutual exclusion. The critical regions are implemented using synchronized objects (for example lockObj in the program sequence above) that perform the functionality of binary semaphores. The information in the database files of the application is *mirrored*. The application keeps duplicate copies of those files. This prevents loosing information due to hardware or software failures that could determine the application abort.
The client part of the application is realized as a Java applet. The user interface was implemented using the Abstract Windowing Toolkit of the Java Development Kit (JDK 1.2). It uses a card-layout (see Athanasiu (2000) and Lemay (1998)). The visualization of the current resource status is done in graphic mode (see Figure 1).

Conclusions

The application presented in this paper was tested in a real environment and assured an improved, more accurate management of the time/classrooms resource at the Faculty of Control Systems and Computer Science at University POLITEHNICA of Bucharest. Without allowing the complete automation of the time-table building process, the usage of this distributed application diminished considerably the effort spent by the teaching assistants assigned to carry out the students time-tables.

References

Anuff, E. (1996) *Java Source Book*, John Wiley & Sons, New York, ISBN 0-471-14859-8

Athanasiu, I. Costinescu, B. Drãgoi, O. Popovici, I. and V. Gãburici (2000) *Limbajul Java, o perspectivã pragmaticã*, Computer Libris AGORA s.r.l., Cluj-Napoca, ISBN 973-97515-4-7

Lemay L. and R. Cadenhead (1998) *Java 2 plate-forme*, CampusPress France, Paris, ISBN 2-7440-0644-0

Mahmoud Q. H. (1999) *Distributed Programming with Java*, Manning, Greenwich, ISBN 1-884777-65-1

Silberschatz, A. Galvin, P. and G. Gagne (2001) *Applied Operating System Concepts*, John Wiley & Sons Inc., New York, ISBN 0-471-36508-4.