CREATIVE MATH. & INF.
17 (2008), 110 - 114

An evaluation of the multithreading benefits for a network scan application

OVIDIU COSMA

ABSTRACT. The network scanning applications are useful for building the search engines databases, and for establishing the security level of a computer network. Such an application usually searches for services through a sequential detection process. This process can be time consuming because each try is followed by a waiting interval, in which the answer from the server is expected. The process can be speeded up by shorting this waiting period, or by reducing the number of retries. Both approaches affect the reliability of the scanning process. This article evaluates a different approach: speeding up the search process by multithreading.

1. INTRODUCTION

A network scan application searches for the servers that offer certain services. Usually the search is performed in a certain list of IP addresses. For this purpose the scanning application systematically sends connection requests to the IP addresses in the scan list, on certain port numbers, and then waits a settled amount of time for the answer from the server. If the answer does not arrive in time, the application can decide to retry the operation for one or more times.

The present article tries to evaluate the increase in a network scan application efficiency that can be achieved by multithreading.

If the scanning application examines more IP addresses [1] in parallel, than better performances can be obtained, because the waiting periods in which the application does not perform useful operations can be avoided.

2. THE SCANNING APPLICATION

The scanning application is based on a thread [2] that sends a certain number of connection requests to a server application. The IP address and the port number at which the server waits for the connection requests are supplied by the module that creates the thread.

The source code of the thread class is presented in Listing 1.

The parameters of the constructor specify the server address, the port number, the number of connection retries, and three *List* objects [4] that hold all the checked IP addresses, the ones that contain active servers and the invalid ones.

The run method describes the operations of the scanning thread. At the beginning, the searched IP address is added to the list of checked addresses. Then follows a loop in which certain connection requests are performed. In case of success, the loop is broken and the IP is added to the list of valid addresses. The end of the

Received: 11.09.2007. In revised form: 18.11.2007.
2000 *Mathematics Subject Classification.* 94A99.
Key words and phrases. *Network scan.*

loop is reached in case of failure, and the IP is added to the list of invalid addresses. A successful connection with the server corresponds to the successful creation of the *Socket* object [3], with the appropriate parameters for the constructor. In case of failure, an *IOException* is thrown.

The *threadNo* variable decremented in the end offers a convenient way to detect the end of the scan operation, which corresponds to zero *CheckThread* threads active in memory.

The CheckThread class

```
import java.awt.*;
import java.io.*;
import java.net.*;

class CheckThread extends Thread{
    String ip;
    int port, retry;
    List success, check, invalid;

    CheckThread(String ip, int retry, List success, List check,
    List invalid, int port){
        this.ip=ip;
        this.retry=retry;
        this.success=success;
        this.check=check;
        this.invalid=invalid;
        this.port=port;
    }

    public void run(){
        boolean ok=false;
        check.add(ip);
        for(int i=0; i<retry; i++){
            try{
                Socket s=new Socket(ip, port);
            }catch(IOException e){
                continue;
            }
            success.add(ip);
            ok=true;
            break;
        }
        if(!ok)
            invalid.add(ip);
        synchronized(NetScan.ready){
            NetScan.threadNo--;
            NetScan.ready.notify();
        }
    }
}
```

Listing 1. The CheckThread class

The main class builds the user interface and launches the network scan by maintaining the number of threads selected by the user. The scanning threads managing process is presented in L isting 2. Launching of new threads is stopped when the total number of threads reaches *maxThreads*, which is selected by the user, and restarted when one of them finishes operation. This way, the total number of threads is maintained most of the time close to the limit imposed by the user.

For stopping the thread creation process the main thread is suspended, and it will be resumed when the next scanning thread finishes operation.

The user interface is presented in Figure 1. In the end, after the last thread finishes operation, the searching time is computed and displayed.

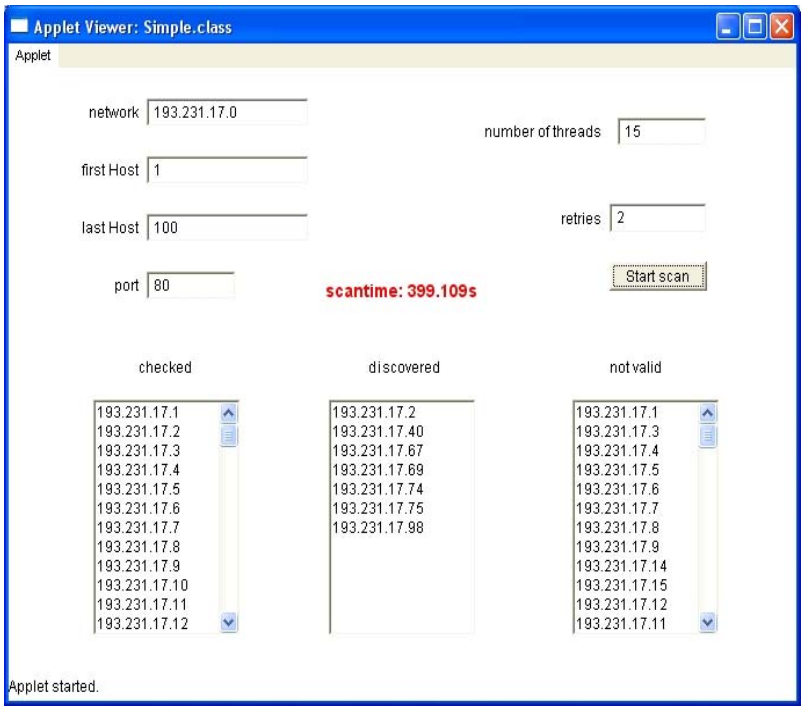

Figure 1. The user interface

3. CONCLUSIONS

The application was tested on computer equipped with an Intel Celeron processor, and the Windows XP operating system. The next graph presents the scanning time for 50 and 100 IP addresses. For each address a single port was tested, and the number of retries was 2. A significant gain in performance could be obtained by increasing the number of threads from 2 to 15. Beyond this value no significant benefits could be obtained, so a number of 15 threads are sufficient. By increasing the number of threads from 2 to 15, the scanning time dropped by approximately 10 times.

```
managing the scanning threads
...
    startScan=new Date().getTime();
    for(int i=firstHost; i<=lastHost; i++){
        synchronized(ready){
            threadNo++;
            while(threadNo==maxThreads)
                try{
                    ready.wait();
                }catch(InterruptedException e){}
        }
        new CheckThread(ipAddress+i, retries, listDiscovered,
        listChecked, listNotValid, portNumber).start();
    }
    synchronized(ready){
        while(threadNo>0)
            try{
                ready.wait();
            }catch(InterruptedException e){}
    }
    System.out.println("gata");
    stopScan=new Date().getTime();
    scanTime.setText("scantime: "+(stopScan-startScan)/1000.+"s");
...
```

Listing 2. Managing the scanning threads

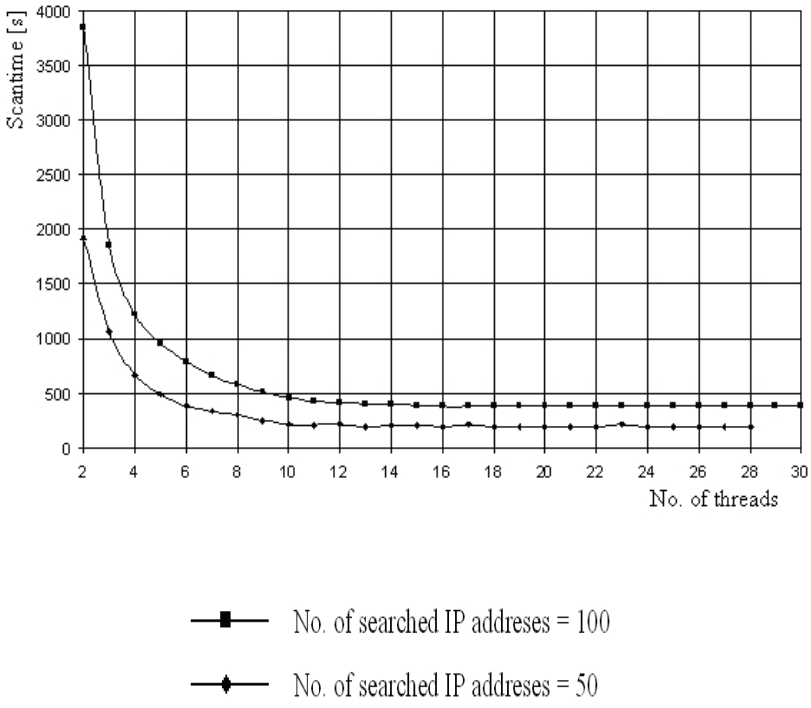

Figure 2. Experimental results

REFERENCES

[1] Andrew, S. Tanenbaum, *Computer Networks*, Prentice Hall, 1989
[2] Ian, Darwin, Java Cookbook, O'Reilly, 2001 e
[3] Elliotte, Rusty, Harold, *Java Network Programming*, O'Reilly, 2004
[4] Java Platform, Standard Edition 6, API Specification, http://java.sun.com/javase/6/docs/api/

NORTH UNIVERSITY OF BAIA MARE
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE
VICTORIEI 76,
430122 BAIA MARE, ROMANIA
E-mail address: o.cosma@rdslink.ro