STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLVI**, Number 2, 2001

AN EFFICIENCY COMPARISON OF DIFFERENT JAVA TECHNOLOGIES

FLORIAN MIRCEA BOIAN

Abstract. Java and related technologies are very used for distributed applications today. In this moment, there are many Java technologies using the client-server paradigm. Among these, the following are the most important: Common Gateway Interface (CGI) [4,7], Servlets/JSP [1,10,12], JavaSpaces [1,5,6] and Enterprise Java Beans (EJB) [1,2,8]. To choose one of these for solving a problem (to implement a distributed application), the performances, rapid development and robustness are important criteria. In this paper, we present a run-time performance comparison between these technologies.

1. Introduction

Platform independence is an important argument for using Java technologies instead others, (for example Microsoft's) distributed technologies. Each of the CGI, Servlet, JavaSpaces and EJB technologies has specific characteristics and implementation difficulties. Theoretically, they are, equivalent: every application implementation using a technology, from one of above, can be (theoretically) implemented using any of the other three technologies.

Of course, from practical point of view there are significant differences. The design and coding effort is relatively reduced for CGI, a little bit more significant for Servlet, a medium one for JavaSpaces and quite impressive for EJB. Taking into account the robustness, security and reliability, we have a reverse order: in the top is EJB; on last position is CGI. It's a difficult, and almost an impossible task, to analyse an compare these technologies in a global and unitary manner. Our opinion is that one has to solve and implement a (some) application(s) using each of the above technologies. Then, make a comparison and behavior analysis of the implementations vis--vis to a (some) criterion(s).

In the present paper, we solve a unique problem: a counter. Four implementations were made: CGI, Servlet, JavaSpaces and EJB. For these implementations, we analyse a single criterion: run-time aspect, both in the server part and client part.

In the next section, the test problem is presented. In the following four sections, the specific architecture and particularities for each implementation are presented.

2000 *Mathematics Subject Classification.* 68M14.

1998 *CR Categories and Descriptors.* C.2.4 [**Computer Systems Organizations**]: Computer-Communication Networks – *Distributed Systems*.

In the last section, some numerical results, comparison between these implementations and some conclusions are presented.

2. The experiment problem: a counter

Most programs for Internet applications are designed using the client / server paradigms and their extensions [3,9]. For our experiments, we use some counter implementations. Particularly, in Internet there are many counters. For example, many Web pages have counters for measuring the total number of accesses to them. The systems for voting popularity pages are based also to counters.

For our purpose, a counter is a pair of the following form:

`(name, value),`

where `name` is a string – the name of the counter – and `value` is the value of the counter – a positive integer.

There are two operations, defined as follows (like Java method prototypes):

```
void counterInit(String name, int initValue);
```

```
int counterAccess(String name);
```

`counterInit` creates the counter name and sets its initial value to `intValue`. `counterAccess` increments, for the counter name, the current value and returns it. Usually, the arithmetic overflow is ignored.

The distributed counter architecture is presented in Figure 1.

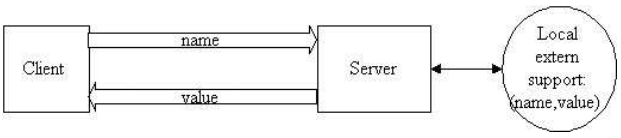

Figure 1. Distributed counter

We adapt the counter implementation so that we obtain the run-times per operation, for client and for server. In the first step, the client invokes the server to create the counter. For our goals, the `counterAccess` returns the run-time – in milliseconds.

The algorithm for the main step, in both the client and server, is presented in Table 1.

The experiment, for each technology, uses two hosts, the same for all technologies: one for the client – only one – and the other for the server. The above algorithms run, without interrupt and without sleeps, waits and so on, for few hours. The connection between hosts was made directly, without any gateways, proxy-s or other related entities. The hosts were run only necessaries services for the experiment.

From time to time, the client saves on a local file, the following information (times are in milliseconds):

- average server time for a `counterAccess`;

TABLE 1

Client	Server

`for ( ; ; ) { // loop forever`
 	
`  ti = theCurrentTime`
`  connect to server`
 	
`  ts = counterAccess`
 	
`  close connection`
`  tf = theCurrentTime`
 	
`  use ti, ts, tf for statistics`
`  save, from time to time`
`  the intermediate statistics`
`  on a local file`
 	
`}`
	`for ( ; ; ) { // loop forever`
	
	`  wait for connection`
	`  ti = theCurrentTime`
	
	`  open access from counter`
	`  access the counter from the`
	`  external support`
	`  increment the value`
	`  save the counter to this`
	`  external support`
	`  close access to counter`
	
	`  tf = theCurrentTime`
	`  return (tf - ti)`
	
	`}`

- average client time for a `counterAccess`;
- maximum server time for a `counterAccess`;
- maximum client time for a `counterAccess`;
- minimum server time for a `counterAccess`;
- minimum client time for a `counterAccess`;
- total server time spend for all `counterAccess`;
- total client time spend for all `counterAccess`;
- total connected time for client: the above time plus the time spent for computed statistics and save intermediate results;
- total number of connections.

The code sources used in the experiment can be accessed at the home page of the author: `http://www.cs.ubbcluj.ro/~florin`. For a uniform approach, and independently of technology, the clients are Java standalone applications, of course, very similar.

The client runs under Windows 2000 Pro, using an AMD K7 (Athlon) at 1GHz, with 512RAM. The server runs under Linux RedHat 7.2, also an AMD K7 (Athlon) at 1GHz, with 512RAM.

3. CGI: FEATURES AND SPECIFICS

The Common Gateway Interface (CGI) [4,7] is a standard for interfacing external applications with information servers, such as HTTP or Web servers. A plain HTML document that the Web daemon retrieves is static, which means it exists in a constant state: a text file that doesn't change. A CGI program, on the other hand, is executed in real-time, so that it can output dynamic information. Since

a CGI program is executable, it is basically the equivalent of letting the world run a program on your system, which isn't the safest thing to do. Therefore, there are some security precautions that need to be implemented when it comes to using CGI programs. Probably the one that will affect the typical Web user the most is the fact that CGI programs need to reside in a special directory, so that the Web server knows to execute the program rather than just display it to the browser. This directory is usually under direct control of the webmaster, prohibiting the average user from creating CGI programs. Interested reader can find details about CGI in [4,7].

For our experiment, the CGI architecture is presented in Figure 2.

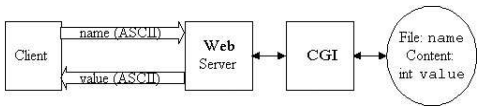

FIGURE 2. CGI architecture

The java standalone client sends to the CGI, using `URLConnection`, the name of the file using the POST method.

The CGI program was developed in ANSI C. For each connection, it opens the (binary) file name having only 4 bytes, in fact an int. Then it reads the value, increments it, rewrites it again and then close the file.

The response is a Content-type: `text/plain` and is the server time, in ASCII form.

4. Servlet: features and specifics

Servlets are the answer of Java technology's to CGI programming [1,10,11,12]. They are programs that run on a Web server and build Web pages. Java servlets are more efficient, easier to use, more powerful, more portable, and cheaper than traditional CGI and than many alternative CGI-like technologies.

As known, for using servlets, a servlet container is necessary. We use the Tomcat container - today a reference servlet container (and free distributable and license free).

For our purposes, only a simple architecture is necessary, so we use only the web facilities offered by the Tomcat 4.x version. Of course, for an important and consistent application, the interfce between Tomcat and Apache Web server must be use.

For our experiment, the Servlet architecture is presented in Figure 3.

The java standalone client sends to the servlet, using `URLConnection`, the name of the file using the POST.

The servlet implements the doPost method. Using the `HttpServletRequest` parameter, the name of the counter is obtained, reading an object String. For each connection, as in the case of CGI, does it open the (binary) file name having only 4 bytes, in fact an `int`. Then reads the value, increments, rewrites again and then closes the file.

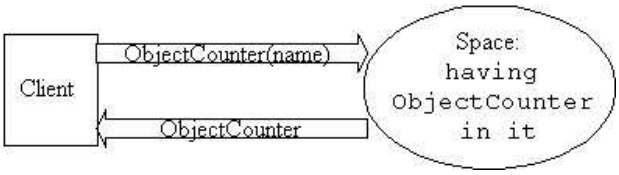

FIGURE 3. The servlet architecture

The response is a Content-type: `application/octet-stream`. It contains the serialization form for an `Integer` object, having the server time in it.

5. JavaSpaces: features and specifics

An important class of distributed applications is based on the JINI technology. JINI is the Sun's solution for creating common, everyday, networking appliances that just "plug and work". Fundamentally, JINI is using in an extensive manner the RMI (Remote Method Invocation) [1,12] Java technology. A programming model/infrastructure JINI enables building/deploying of distributed systems organized as *federations of services*. A service can be a hardware or software component, a common channel, a user, a disk drive which can offer a storage service and so on. Once part of a federation, a service can be used by other services or clients

An important particular application of JINI is JavaSpaces [5].

A JavaSpaces server is called *space*, and holds *entries* (typed groups of objects) in it. A distributed Java Spaces application has a space, and a lot of clients that access this space. There are three main operations over space:

write: put an object in space using a special Entry object - a container for the objects from space;

read: get a copy for an object from space. Look it up using *template* entries, using fields with values (exact matching) and null for wildcards. The object is found by associative methods. If such an object is not in the space, the client waits until such object put in the appearance in the spaces.

take: similar with read, find an object from space and move the object from space into the client program.

Therefore, an elegant paradigm for distributed programming and concurrent programming is provided. All operations are transaction security. Entries written to a space are leased (using JINI leasing). For short, a JavaSpaces acts as a "shared memory" for distributed processes. We can preserve any kinds of objects in space, with elaborate "data structures" in them.

For our experiment, the java source of the objects from the spaces is defined as follows:

```
public class ObjectCounter implements Entry {
   public String name;
   public Integer counter;
```

```
    public ObjectCounter() {
    }//ObjectCounter.ObjectCounter
    public ObjectCounter(String name) {
       this.name = name;
    }//ObjectCounter.ObjectCounter
    public ObjectCounter(String name, int counter) {
       this.name = name;
       this.counter = new Integer(counter);
    }//ObjectCounter.ObjectCounter
    public Integer increment() {
       this.counter = new Integer(counter.intValue() + 1);
       return counter;
    }//ObjectCounter.increment
 }//ObjectCounter
```

The `ObjectCounter()` is mandatory for JavaSpaces Technology. The method `ObjectCounter(String name, int counter)` is used for the init part and the method `ObjectCounter(String name)` is repeatedly used for *take* and *rewrite* objects in the space. The JavaSpaces architecture for this application is presented in Figure 4.

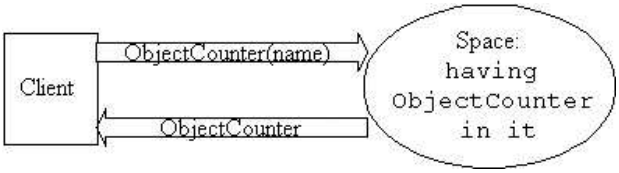

FIGURE 4. The JavaSpaces architecture

The time for a client access includes the operations `setSecurityManager`, construct a `LookupLocator` object, define a `ServiceRegistrar` object and `lookup` the space object. (For details, see [1,6]).

After that, we consider part of the server time, and its current standalone actions are:

```
template = new ObiectContor(NameOfTheCounter);
counter = (ObjectCounter)space.take(template, null, Long.MAX_VALUE);
Integer i = counter.increment();
space.write(counter, null, Lease.FOREVER);
```

6. EJB: FEATURES AND SPECIFICS

Enterprise JavaBeans (EJB) [2,8,12] are a container-based component architecture that allow you to easily create secure, scalable and transactional enterprise applications. Developed as session beans, entity beans, or message-driven beans, EJBs are the critical business objects in any J2EE application. The 2.0 version of the specification for EJB introduces important improvements to the bean-managed (BMP) and container-managed (CPM) models for entity persistence.

Our experiment uses the 2.0 CPM model for the counter implementation. The important parts of the Java source excerpt are given below. Firstly, the remote interface is:

```
import javax.ejb.*;
import java.rmi.*;
public interface Counter extends EJBObject {
   int increment() throws RemoteException, FinderException;
}//Counter
```

The home interface is:

```
import java.io.*;
import java.rmi.*;
import javax.ejb.*;
public interface CounterHome extends EJBHome {
   Counter create(String name, int value)
      throws RemoteException, CreateException;
   Counter findByPrimaryKey(String name)
      throws RemoteException, FinderException ;
}//CounterHome
```

The main part of the Entity bean is:

```
import javax.ejb.*;
import java.rmi.*;
public abstract class CounterBean implements EntityBean {
   private EntityContext context;

   public abstract String getName();
   public abstract void setName(String name);
   public abstract int getValue();
   public abstract void setValue(int value);

   public int increment() throws RemoteException, FinderException{
      long ti = System.currentTimeMillis();
      int i = getValue();
      setValue(i+1);
      return((int)(System.currentTimeMillis()-ti));
   }//CounterBean.increment

 - - - - - - - - - - - - - -

}//CounterBean
```

The main part of the standalone client is:

```
 - - - - - - - - - - - - - -
Context initial = new InitialContext(env);
Object ref = initial.lookup("aliasCounter");
CounterHome counterHome = (CounterHome) PortableRemoteObject.narrow(
                                              ref, CounterHome.class);
Counter counter = counterHome.findByPrimaryKey("nameOfCounter");
ts = (long)counter.increment();
 - - - - - - - - - - - - - -
```

The EJB architecture for this application is presented in Figure 5.

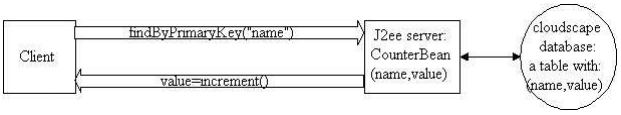

FIGURE 5. The EJB architecture

7. TIME COMPARISONS

For comparing these technologies, from the run-time point of view, we made many experiments. At `http://www.cs.ubbcluj.ro/~florin` there is an archive with files having results and program sources. CGI, Servlet and JavaSpaces, codes run in the same conditions, for about 8 hours. EJB was experiment only for 1 hour.

In the following table, we present the run-times after run about 1 minute, 10 minutes, 30 minutes, 1 hour, 4 hours and 8 hours. After each 1000 tests, the client programs verify if the moments of saving results appeared or not. For this reasons, the times for total connections have some differences between our intention to save the results and the real moments to save.

Min.	Parameter	CGI	Servlet	JavaSpaces	EJB
1	TotalClient	132901	130338	151057	256569(43171)
	TotalServer	9671	93260	72045	20(22)
	MinimumClient	10	0	30	161(80)
	MinimumServer	0	0	20	0(0)
	MaximumClient	410	231	540	1092(571)
	MaximumServer	398	153	160	10(4)
	AverageClient	14	14	50	256(43)
	AverageServer	1	10	24	0(0)
	NumberOfTests	9000	9000	3000	1000(400)
	TotalConnectClient	132941	130398	151067	256579(43612)
10	TotalClient	663214	670645	694078	705275
	TotalServer	65752	471949	311208	60
	MinimumClient	10	0	30	150
	MinimumServer	0	0	10	0
	MaximumClient	611	250	2974	1092
	MaximumServer	597	173	230	10
	AverageClient	15	13	53	235
	AverageServer	1	9	23	0
	NumberOfTests	44000	49000	13000	3000
	TotalConnectClient	663354	670805	694178	705305

30	TotalClient	1868597	1863150	1892922	2068375
	TotalServer	266662	1310745	855115	201
	MinimumClient	10	0	30	150
	MinimumServer	0	0	10	0
	MaximumClient	881	251	2974	1742
	MaximumServer	857	246	390	11
	AverageClient	15	13	54	229
	AverageServer	2	9	24	0
	NumberOfTests	117000	136000	35000	9000
	TotalConnectClient	1868817	1863500	1893102	2068465
60	TotalClient	3672722	3670319	3682746	3745356
	TotalServer	433542	2611931	1656224	351
	MinimumClient	10	0	30	150
	MinimumServer	0	0	10	0
	MaximumClient	982	321	3025	2674
	MaximumServer	961	311	390	11
	AverageClient	15	14	54	234
	AverageServer	1	10	24	0
	NumberOfTests	234000	260000	67000	16000
	TotalConnectClient	3673022	3670959	3682996	3745536
240	TotalClient	14460504	14458049	14475715	
	TotalServer	2015054	9953004	6532319	
	MinimumClient	10	0	30	
	MinimumServer	0	0	10	
	MaximumClient	1001	401	3085	
	MaximumServer	984	311	421	
	AverageClient	16	13	54	
	AverageServer	2	8	24	
	NumberOfTests	903000	1111000	264000	
	TotalConnectClient	14461685	14460173	14476456	
480	TotalClient	28558686	22400298	28861269	
	TotalServer	3223970	14725660	12946711	
	MinimumClient	10	0	30	
	MinimumServer	0	0	10	
	MaximumClient	1001	6399	3085	
	MaximumServer	984	6390	3034	
	AverageClient	15	13	54	
	AverageServer	1	9	24	
	NumberOfTests	1839000	1625000	527000	
	TotalConnectClient	28865617	29642314	28862612	

However, EJB performance is very dependent on the underlying configuration. For example, informal tests (`http://www.JBoss.org`) show that on the same PC box, it can run twice as fast under Windows 2000 / Sun JVM than under Linux 2.2 / Sun JVM. Linux users probably already know that linux does not support real

threads. Under heavy load, JBoss will for example crash with 200 concurrent users under linux, whereas it can handle 1000 of them on the same box with Windows 2000. Of course, if you use Apache or Jetty in front of JBoss to handle the thread pooling, this will not be a problem.

In our experiment, EJB randomly crashes on Linux, between 100 to 400 tests. In the first cell from EJB in the above table, we write in brackets, our results after 400 tests.

The main parameter in which we are interested is the *average time client*. **From this point of view, our conclusion is**:

- For this kind of applications, the Servlet technology is optimal. We consider that 13 millisecond is a good average time client.
- The CGI technology have a nearly performance, 15 millisecond is a value nearly of the Servlet technology. The difference is due to the fact that CGI uses much more system resources than Servlet [10].
- JavaSpaces is an elegant solution, but its performance is about 3 times sluggish than Servlet and CGI technologies.
- EJB technology is a robust one, but only if it use a professional EJB server (as JBoss or BEA for example) with a professional operating system (as Solaris, for example).

Thus, for simple distributed applications, with very frequent rate of use but with simple security restrictions we recommend to the use a Servlet or CGI technologies. JavaSpaces and EJB are recommended to be use only for large applications, with a medium frequency of use and with high security and transactional restrictions.

References

[1] Ayers D. et. al. *Professional Java Server Programming* Wrox Press, 1999
[2] Bodoff S. et.al. *The J2EE Tutorial*. Sun Microsistems, 2001
[3] Boian. F.M. *Distributed programming in Internet* (Romanian) Blue ed. Cluj, Romania, 1998
[4] Breedlove et. al. *Web Programming Unleashed*. Sams.het, 1996, http://sams.mcp.com
[5] Edwards W.K *Core Jini*. Prentice Hall, 1999
[6] Halter S.L. *JavaSpaces Example by Example*. Prentice Hall, 2002, http://www.phptr.com
[7] Kim E. et. al. *CGI Programming Unleashed*. Sams.het, 1996, http://sams.mcp.com
[8] Roman E. *Mastering Enterprinse JavaBeans and the Java 2 platform*Willey, 1999
[9] Umar A. *Object Oriented Client / Server Internet Environments*. Prentice Hall, 1997
[10] * * * http://http://www.coreservlets.com
[11] * * * http://jakarta.apache.org
[12] * * * http://java.sun.com/

University "Babeş-Bolyai", Cluj-Napoca, Romania
E-mail address: florin@cs.ubbcluj.ro