Pool of Processors on the Web

Denivaldo Lopes[1], Osvaldo Saavedra[2], Zair Abdelouahab[3]
[1,2,3] Federal University of Maranhão
Campus Bacanga, São Luís, MA, 65080-040, Brazil
{[1]dlopes@dee.ufma.br, [2]osvaldo@dee.ufma.br, [3]zair@dee.ufma.br }

Abstract—
This paper presents ESOW (An Environment of Shared Object for Web) to support parallel and distributed programming on Web or Intranet. The environment allows that computers with low computational resources take advantage of idle computers present on the Web. ESOW is a Pool of available computers to share two basic resources: processor and memory. The main entities in this environment are passive and active objects. Passive objects are lists, stacks and queues which store others objects defined by users. While active objects are processes. The proposed environment implements load balancing and tolerance to fault. Some tests show the ESOW power.

Keywords — **Web, Load Balancing, Distributed and Shared Objects, Tolerance to fault**

I. Introduction

The invention of computer caused a deep impact in human life. In the past years all science areas have its evolution dependent of computer progress and the behaviour of people was modified. The next revolution will not just be technological, but cultural. The Internet will not bring only technological or behavioural changes, but will implicate in a coalition of the cultures. Perhaps this takes some decades to happen. But with certainty, it is already a reality the use of the Internet as the largest multicomputer already invented by man.

In this context, it appeared the possibility of taking advantage of available resources nodes of Internet and Intranet. But it is not simple to program in distributed systems.

This paper shows an environment with shared objects for Web (ESOW) which is an evolution of other work [ABD 00]. Objects may be passive or active entities capable to control the complexity of distributed and parallel systems. Passive objects are stored in data structures such as lists, queues and stacks which are shared among nodes of the Web.

The paper is organised in five sections. The first section introduces the paper. The second section presents an overview on distributed systems. The third section shows the developed environment, detaching its architecture, operation and any results obtained. The fourth section presents a comparison among ESOW and other environments. Finally, the last section outlines a conclusion on the work.

II. Background

A. Distributed systems

In the last decades several powerful parallel architectures and high-speed networks were invented always looking for high performance. It was invented two architecture: multiprocessor and multicomputer. Multiprocessor systems exhibit a semantics of predictable performance, but possesses a very complex hardware; they are difficult to construct, are inherently not scalable and share a space of common address. Distributed systems belong to the multicomputer systems, in that each processor is associated to its own memory, and is connected to another processor/memory through a high-speed network. These systems are easy to built and are scalable. The combination multiprocessor and multicomputer system resulted in DSM (Distributed Shared Memory) systems that has the benefits of this two systems. A Software for Distributed Shared Memory (SDSM) system is an evolution of DSM that simplifies programming on homogeneous platforms by presenting the illusion of shared memory [STE 99].

Another strategy is to use the concept of shared object (SO), that supplies a common address space shared by all processes of the systems. The processes interact with the objects through message passing. As an extension to SO, DSO (distributed shared objects) has as foundation the physical distribution of the objects [TAN 99]. Objects in DSO are distributed physically among the machines of the network, but they are seen as one and indivisible entity.

The DOT (Distributed Object Technology) is formed by three technologies: object technology, distribution technology and Web technology. The important aspect of DOT is the interconnection that is implemented though a middleware. Among the most common, they stand out Open Software Foundation's - Distributed Computing Environment (OSF/DCE), Common Request Broker Architecture (CORBA) [OMG 99] and Java Remote Method Invocation (RMI) of Sun [MAS 99].

The language Java provide many benefits such as simplicity, oriented to object, secure, neutral and portable. Its use with CORBA and integration of RMI over IIOP (Internet Inter-ORB Protocol) with other technologies such as CORBA, DCOM [RAP 00]. Many projects are using

Java as base to develop an extended language to address the complexity of distributed/parallel environments, such as Ninflet [TAK 98], Javalin [CHR 97] and Charllote [BAR 96].

III. Environment of Shared Objects for Web

A. Overview

ESOW is an environment for developing distributed and parallel application based on the conception of shared objects. The adopted computational model supplies users with three collections of objects classes SOC, DSOC and ASMOC:

SOC (Shared Object Classes): is a collection of classes that allows the implementation of data structures as queue, stacks and lists. These data structures store objects centralized physically in one machine and logically shared by all the other machines of the environment. The objects stored inside of the data structures are passive, in other words, does not have processes inside in the object. Therefore, it just possesses the inter-object concurrency.

DSOC (Distributed Shared Object Classes): is similar to the SOC, but there is a difference in that it allows the objects to be distributed physically among the machines that form the environment. Thus, the data structures such as list, queue and stacks are physically distributed.

ASMOC (Active Shared Mobile and Object Classes): is a collection of classes that uses the notion of active and mobile objects. The objects possesses only one process and can be migrated in agreement with the algorithm of load balancing of ESOW. The migration of a active objects is possible in ESOW, but the process is restarted from the beginning or restarted from a predetermined point by programmer. Active objects can be shared by any other machines beyond the one in which it was originated. Unlike traditional threads in Java, active objects of our environment are not destroyed at the end of execution, they stay alive until an explicit command is issued ordering their destruction. That is important, because the results of processing active objects are available for other processes.

ESOW was developed in the language Java in the form of a library and a Kernel, because Java is a quite attractive language for metacomputing applications [NIE 99]. The library of ESOW should be used by an application or applet. To give support to programming in the Web, the notion of an architecture multi-tier is used. In the case of programs for Web, the clients runs an applet that creates and uses shared objects through a mediator (Broker). The Broker is responsible for doing the creation, destruction, location, reading and writing on object instanced from the collections of classes SOC, DSOC and ASMOC. In the case of an application, the library can communicate directly with a local Kernel. The Figure 1 show one application of ESOW.

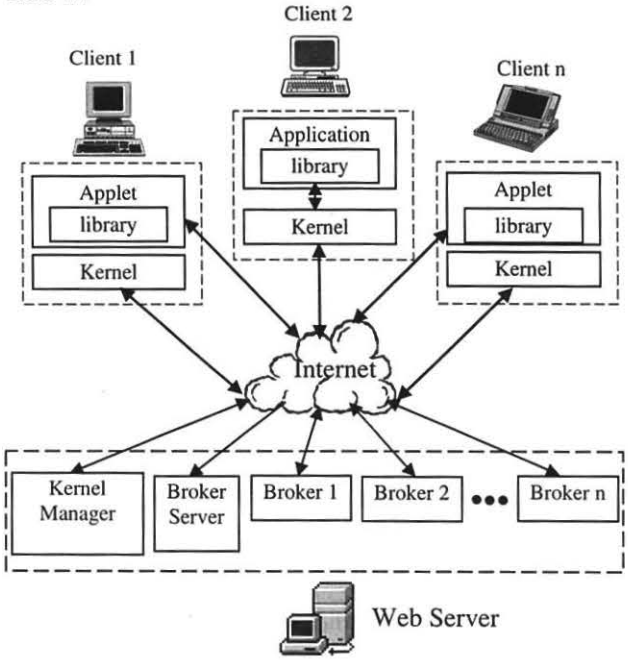

Fig. 1 Application of ESOW on the Web

ESOW use an algorithm of hybrid load balancing (static and dynamic) to improve the allocation of the objects in order to increase the performance of the system. The characteristic static because the load balancing occur when an object is created and dynamic because in time of execution is possible that the object migrates.

B. Kernel

There are two types of Kernels: With a manager and without a manager. A kernel without a manager has the functionality of controlling the life cycle of an object and to monitor the computational resources in its node; Kernel with a manager has more functionalities and should run on a Web Server on the same node or on another dedicated machine. Both Kernels are similar. We illustrate only a Kernel with a manager in Figure 2.

The Kernel without a manager is formed by the structure server, monitor of load, monitor of persistence and fault, specialised server (passive and active objects) and access to RMI. The structure server is responsible for the creation and destruction of the specialised servers. The monitor of load is a process responsible for obtaining the parameters necessaries for load evaluation. The monitor of persistence and fault (MPF) detect in the initialisation of Kernel if occurred crash and restart objets from persistent state.

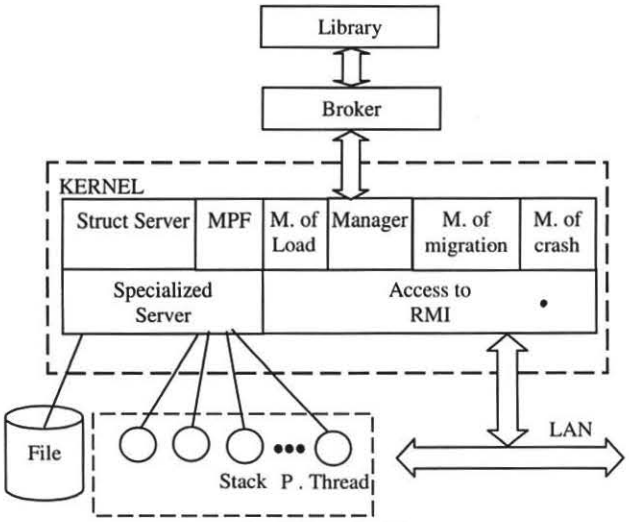

Fig. 2 Kernel with a manager

The Kernel with a manager has the same functions of the kernel without a manager and also has the task of nominating the objects, indicating on which nodes the objects should be allocated and recuperating the defective nodes.

An object in ESOW possesses two names, a logical that is given by the user and another physical name that is given by the system. With the logical name, the objects are shared by the processes, while the physical name is used by the Kernel to control its location, replication, persistence and its life cycle.

The manager maintains information load of the distributed environment. These information are enough for the implementation of the load balancing. The monitor of crash verifies the occurrence of any fault of a node, and make arrangements to substitution of the objects that faulted for its replicas. In Case an object does not have replicas, the clients await until the persistent state is recovered. The migration monitor co-ordinates together with the specialised servers the stages of the migration.

Object Space is implemented by the Java language through RMI. Each machine with a Kernel has its repository, and the sum of all the repositories forms the global repository of ESOW.

The specialised servers implement data structures (lists, queues, stacks, etc.) that have its elements (passive objects) in one node in case it is instantiated from the SOC collection or physically distributed among nodes in case it is instantiated from DSOC. An active object (ASMOC) possesses one internal process and can also be migrated.

ESOW implements security on objects. Each object has a password. ESOW allows a state of an object to be read or altered by applications or applets that know its password. When the library makes the creation or sharing of an object, should be inserted a password that is created by the programmer. That is necessary, because the objects should be shared among authorised processes.

Another important characteristic of ESOW is that it restricts access to computer resources by active objects, such as file, directory, socket, system information.

If a Specialised Server of a physically distributed type needs more memory than its computer can offer, the allocation is accomplished in another machine. This implies that the List, Queue and Stack has its elements distributed physically. A configuration of the environment is illustrated in Figure 3.

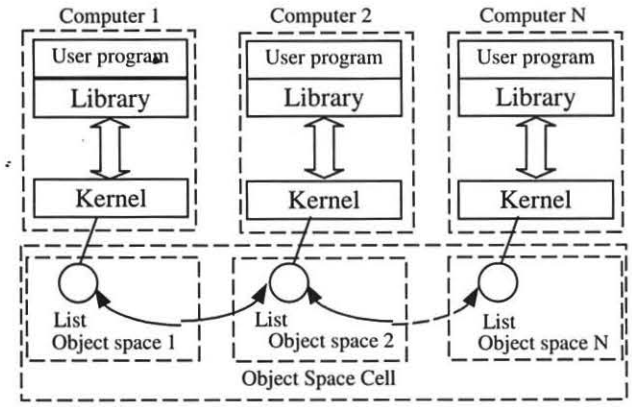

Fig. 3 DSOC Object

The physical distribution of a structure is limited to one cell, but can be accessed by others computers.

C. Load Balancing and Fault Tolerance

Load balancing allows that all resources presents on ESOW to be utilised. A strategy of load balancing involves many characteristics such as: co-operation, control location, initialisation; and properties: static, dynamic, adaptive, etc. The properties that have prominence are static and dynamic load balancing. In static balancing, the distribution of work load is made in the phase of initialisation of the computation [WAT 95]. The dynamic load balancing happens during the computation, implying that migration of processes is done during the time of execution. The hybrid load balancing is the junction of the static and dynamic load balancing.

In this work, the parameters suggested by [CHO 98], [WAT 95], [THI 00] and [RAC 97], have been used to accomplish the load balancing. Such parameters were modified and added to others to implement load balancing. The parameters are:

- Time of creation and migration of an object in certain machine (TC and TM, respectively);
- Object's execution rate (ER);

$$ER = \frac{\text{time of object's execution}}{\sum_{i=1}^{m} i} \times 100$$

- Amount of available memory in the node (AM);
- Object's request concentration rate (Rreq);

$$Rreq = \frac{\text{number of request on this node}}{\sum_{i=1}^{n} \text{number of request on i node}} \times 100$$

- Object's average request waiting time rate (Rwait);

$$Rwait = \frac{\text{waiting time on this node}}{\sum_{i=1}^{n} \text{waiting time on i node}} \times 100$$

- Object's reply concentration rate (Rrep);

$$Rrep = \frac{\text{number of reply on this node}}{\sum_{i=1}^{n} \text{number of reply on i node}} \times 100$$

- Object's processing time (Rproc);

Reserved Time to an objects for execution (Time Slice - TS).

where: n = total number of node
m = number of processes in this node

The algorithm is:

```
// load balancing
// TC_M – TM_M – AM_M – ER_M – Rreq_M – Rrep_M – Rproc_M // – Rwait_M –
TS_M are average values.

load_balance(...)
  for i=1, to n                    // n: amount of nodes
      get and calculate average of TC, TM, AM, ER, Rreq, Rrep,
          Rproc, Rwait, TS
  end for
  if (task == create object)
     best_am [ ]= nodes with AM > size of object
     best_ts [ ]= nodes of best_am [ ] with TS > TS_M
     best_rproc [ ]= nodes of best_ts[ ] with Rproc < Rproc_M
     best_tc[ ]= nodes of best_rproc[ ] with TC < TC_M
     best_node = the node best of best_ tc[ ] with Rreq < Rreq_M
     create object on best_node
   end if
if ( task == migrate )
     bad_ts [ ] = nodes with TS < TS_M
     bad_rproc [ ] = nodes of bad_ts[ ] with Rproc > Rproc_M
     bad_node = bad_rproc[ ] with AM < AM_M
     objects[ ] = objects on bad_node with ER > ER_M
     best_am[ ]= nodes with AM > size of objects[ ]
     best_tm [ ]= nodes of best_am [ ] with TM < TM_M
     best_ts [ ] = nodes of best_tm[ ] with TS > TS_M
     best_rproc[ ]= nodes of best_ ts[ ] with Rproc < Rproc_M
     best_tc[ ]= nodes of best_rproc[ ] with TC < TC_M
     best_node = the best node of best_ tc[ ] with Rreq < Rreq_M
     move objects[] from bad_node to best_node
   end if
end load_balance
```

Several methodologies were created to guarantee a level of tolerance to fault. Among them, stands out replication of the servers and servers with persistent state [KEL 99]. Our proposed environment allows replicated and persistent object model. The objects are distributed physically among the nodes and each fragment of the object is replicated in another machine (copy of the object fragment). In case of both machines are broken (that is, the computer with original object and computers with a copy), the system still stay consistent because it has persistent objects. The reliability level can be configured such as Table I.

TABLE I
Reliability levels

Replication	Persistence	Level	Extremes
0	0	I	without reliability
0	1	II	
1	0	II	
1	1	IV	maximum reliability

It is important to observe that our environment without reliability has a maximum performance but with maximum reliability it has a minimum performance. This occur because the implementation of replication and persistence consumes computational resources. The user is responsible for determining the reliability level that satisfies his needs.

The kernel is configured by users through the ESOW's Console (implemented with JFC). It is possible define and analyse: level of reliability, security, statistical information, search for others kernels in others intranets or internet, etc.

D. Programming with ESOW Library

Programming with shared object in the proposed environment is very simple. The first step consists of instantiating objects (the library SOC, DSOC or ASMOC). If objects are instanced of SOC, DSOC, the user is required to supply the logical name of the object, URL of Web Server in case of applet, should have a clone, should have state persistent. The following code shows the syntax for creation and usage an object of SOC:

```
import java.applet.*;  import java.awt.*;  import structs.applet.*;
import structs.Kernel.*; import java.net.*;  import structs.broker.*;
public class Teste extends Applet implements Runnable {
...
   public void init() {
    URL from= getCodeBase();
   List_soc List = new List_soc("list4",from.getHost(),true,false);
  Stock st= new Stock ();
  List.insert_front(st);
  ....
   Stock  nst= (Stock) List.remove_back();
   }
}
```

And the following code utilise active objects to do multiplication of two arrays:

```
// This code extend Thread_remote belonging ASMOC
import structs.Kernel_asmoc.*;
public class Matrix extends Thread_remote{
 ...
 public void init_remote() {  // user put the code here
   for(int i=0; i < a_rows; i++)
     for(int j=0; j < b_cols; j++)
     {
```

```
      float sum=0;
      for(int k=0; k < a.cols; k++) sum=sum+a.value[i][k] * b.value[k][j];
        }
        c.value[i][j]=sum;
    }
     set_output(c);  // return result
}

import structs.Kernel.*; import structs.Kernel_asmoc.*;
import structs.applet.*;
public class Client_mat {
...
public static void main(String[] args) {
// read matri,  after broke matrix in sub-matrix and put in Thread_remote
Matrix mat1 = new Matrix ("A",line_a/2,row_a,line_b,row_b/2,a_1,b_1);
Matrix mat2 = new Matrix("B",line_a/2,row_a,line_b,row_b/2,a_1,b_2);
Matrix mat3 = new Matrix("C",line_a/2,row_a,line_b,row_b/2,a_2,b_1);
Matrix mat4 = new Matrix("D",line_a/2,row_a,line_b,row_b/2,a_2,b_2);
Pool  thr_env_1 = new Pool(false,mat_1);
Pool  thr_env_2  = new Pool(false,mat_2);
Pool  thr_env_3  = new Pool(false,mat_3);
Pool  thr_env_4 =  new Pool(false,mat_4);
thr_env_1.start();  thr_env_2.start();  thr_env_3.start();  thr_env_4.start();
thr_env_1.join();  thr_env_2.join();  thr_env_3.join();  thr_env_4.join();
float [][] res1= thr_env_1.get_output();
float [][] res2= thr_env_2.get_output();
float [][] res3= thr_env_3.get_output();
float [][] res4= thr_env_4.get_output();
}
```

This code is the same used in section 3.5 in order to test our environment. The use of ASMOC has two main steps. First, it consists of a creating an object instance from Thread_remote that is a process. The Second step consists of creating an environment into the Pool of processors. The instanced object has two parameters. The first is a boolean flag to indicate whether to start or not start the process when the object is created and the second is the process itself . There are other methods and parameter that are not mentioned in this paper, due to space.

IV. TESTS WITH ESOW

The implementation of ESOW has been done using VisualAge1 3.0 and 3.5 of IBM. The system is developed as a library to be linked by Java environment. In our case, we used JDK 1.2.2 for Windows and Linux of Sun Microsystems. The results showed in this paper are better than other results presented in last work [ABD 00], because many modification were made with intention to optimise the system performance. We have done several tests with ESOW and the results were satisfactory. The configuration of our environment is shown in Table II. These machines wasn't dedicated.

TABLE II
Platforms

A	B	C	D
Pentium III	Pentium III	Pentium II	Pentium II
650 MHz	650 MHz	266 MHz	266MHz
10 Mbs *	10 Mbs *	10 Mbs *	10 Mbs *
Win. 98	Linux	Win. 95	Win. 98

* network card

In order to test the proposed environment, we have instanced two lists of the class SOC. The first list was created through an application and the other in an applet downloaded through the browser Netscape Communicator 4.76. The configuration for the tests were similar to Figure 1. So much the application as the applet implemented a program of stock that needed to be distributed. The inserted and removed objects of the lists were instanced from the following class:

```
public class Stock implements Serializable {
        Code code_product;  String name_product;
        C_date date_input, date_output;
        Float value_input, tax, gain, final_value, int_rate; }
```

The class also possesses some methods that were intentionally omitted.

840 elements are inserted in each list and distributed among the four machines (in order to force the physical distribution, we restrict the maximum number of objects that a list could have in one machine - 200 objects).

The values presented in the tables were calculated though the formulas:

$$\overline{T}_{maximum} = \frac{\sum_{i=1}^{n} T_i}{\sum_{i=1}^{n} i} \quad \text{and} \quad \overline{T}_{minimum} \frac{\sum_{i=1}^{n} t_i}{\sum_{i=1}^{n} i}$$

where: T_i – is the maximum value of each test
t_i – is the minimum value of each test
n – is the number of tests

The creation, insertion and removal on list have brought the results presented in Table III.

TABLE III
Evaluation of the Structure List

Parameter of Measure: T – time (ms)			
Type	Task	$\overline{T}$minimum	$\overline{T}$maximum
Application	Creation of a list	110	660
	Insert	<< 1	50
	Remove	<< 1	56
Applet	Creation of a list	853	1430
	Insert	<< 1	241
	Remove	<< 1	263

The values obtained with the applet were greater than those obtained with the application because the applet invokes the Broker and executes in the browser.

It was observed that the migration is an excellent methodology to optimise the use of the network resources (network, Intranet and Internet). In order to test the migration, the performance of each machine was varied through the execution of several processes. Table IV displays the results obtained with an object (SOC and DSOC) that migrated from the machine of origin to another destination machine. Other parameters were also analysed, but in this paper we just show TS (Time Slice). The process of migration is designed in six steps: 1) The

monitor of migration detects one node that has the smaller performance; 2) The monitor of migration queries struct server on this node which objects need to migrate; 3) the monitor asks the manager what is the better node and put the object in state of migration; 4) Then it is created another object instanced from classes SOC, DSOC or ASMOC on a better node; 5) A copy of object is then done ; 6) The original object is destroyed and the table in the manager is updated (the object is placed in state of normality). While one object is in state of migration this object is unavailable for access.

TABLE IV
Migration

Parameter of Measure: $\overline{T}$ – time (ms)				
From		To		$\overline{T}$ (migration)
Plat.	TS	Plat.	TS	
C	5650	B	18630	262
B	3676	D	6744	648
D	5392	A	19048	328
A	4934	C	6528	731

The reliability of the system was evaluated simulating the break of three machines, see table V. Machines B, A and D have objects whose clones were in machines A, D and C, respectively. So much the detection as the recovery of the objects through the clones were transparent for the applications and applets, although it has occurred an overhead.

TABLE V
Recuperation to fault

Change			
From	To	$\overline{T}$minimum(ms)	$\overline{T}$maximum(ms)
B	A	492	788
A	D	513	936
D	C	471	894

To evaluate the tolerance to fault with persistent state, we simulated the crash of machines A and D. The computer A has an object and computer D stored its clone. When ESOW perceived the fault, first it tried to recoup clone, but computer D also had failed. However, the system still could be restored, through the persistent state. Then the system wait to get in operation the machine A. The load time of the OS (Operational System) and the Kernel of ESOW take some minutes to start (2-5 minutes was the observed time). The advantages of ESOW with persistence is reliability and transparency of fault.

The use of active object through ESOW is simple. A test program implemented the multiplication of arrays of parallel/distributed form.

The program made the multiplication of two arrays. Each array was divided in four sub-arrays for multiplication.

The results are shown in Figure 4, where the value obtained with sequential program on PC-266 and PC-650 and parallel program on ESOW.

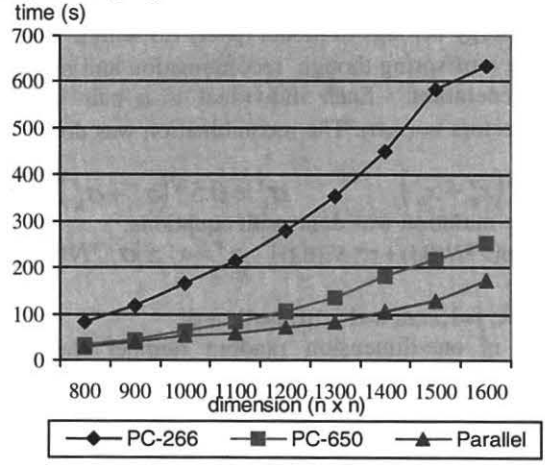

Fig. 4 Execution time

Four computers are used (see table 2). The speedup of the tests was determined with formulas:

$$S(n) = \frac{\text{Execution time using one processor (bad processor)}}{\text{Execuiton time using n processor with ESOW}}$$

$$S'(n) = \frac{\text{Execution time using one processor (best processor)}}{\text{Execuiton time using n processor with ESOW}}$$

In this case, the speedup is shown maintaining the number of processors invariable and the dimension of array variable. This choice was done because we are interested in measuring the behaviour of our environment with much communication. Figure 5 presents the results.

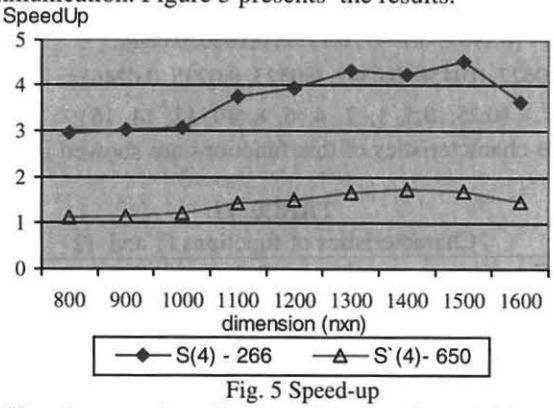

Fig. 5 Speed-up

We observe that the speed-up is done taking as a reference the bad and best machine. This is necessary because our environment is heterogeneous. The values disclose that ESOW with four machines have more potential than one machine only.

It was done a test using objects instanced of SOC and ASMOC classes. It was implemented a parallel evolution strategy program. The evolution strategies are a class of general optimisation algorithms which are applicable to functions that are multimodal, even discontinuous or non

differentiable [YAO 99]. A global minimisation problem can be formalised as a pair (S,f), where $S \subseteq Rn$ is a bounded set on Rn and $f: S \rightarrow R$ is a n-dimensional real-valued function.

Our strategy belongs to model ($\mu+\lambda$)-ES with μ parents to generate λ offspring though recombination and mutation in each generation. Each individual is a pair of real-valueted vectors $v=(x,\sigma)$. The recombination was done with equations:

$$x_i^j = 0.5*\left(x_a^j + x_b^j\right) \qquad \sigma_i^j = 0.5*\left(\sigma_a^j + \sigma_b^j\right)$$

And the mutation was done with equations:

$$\sigma_i' = \sigma_i * \exp(\tau' * N(0,1) + \tau * N_i(0,1)) \quad x_i' = x_i \pm |\sigma_i' * N(0,1)|$$

where:

i=1,..., λ; j=1,...,n; a=1,...,μ; b=1,..., μ

N(0,1) is one-dimension random number though a Cauchy density function. It is the same for all elements of individuals. Ni(0,1) is a new number for each element i.

The number τ and τ' was used by [YAO 99]:

$$\tau = \left(\sqrt{2*n}\right)^{-1} \text{ and } \tau = \left(\sqrt[4]{4*n}\right)^{-1}$$

The signal $\pm$ denotes a random choose between + and –.

Observe that a mutation of offspring the variance will be different to each generation (self-adaptation).

Yao and Liu [YAO 99] presents 23 well-know functions denoted of benchmarks functions in study of evolution strategies. We implement two functions to test our program based in ESOW. The functions are:

$$f_1(x) = \sum_{i=1}^{n} x_i^2 \qquad f_2(x) = \sum_{i=1}^{11}\left[a_i - \frac{x_1\left(b_i^2 + b_i x_2\right)}{b_i^2 + b_i x_3 + x_4}\right]^2$$

where:

$a_i = \{0.1957, 0.1947, 0.1735, 0.1600, 0.0844,$
$0.0627, 0.0456, 0.0342, 0.0323, 0.0235, 0.0246\};$

$b_i^{-1} = \{0.25, 0.5, 1, 2, 4, 6, 8, 10, 12, 14, 16\};$

The characteristics of this functions are showed in table VI.

TABLE VI
Characteristics of functions f1 and f2

Function	n	S	fmin(x)
f1	30	[-100,100]n	0
f2	4	$[-5,5]^n$	0.0003075

where:

n: a dimension; S: a domain; fmin: a global minimum

The results are showed in table VII and table VIII, the values obtained for Yao are shown too (except time, because Yao was concentrated in optimal values and not in performance of computers). In our work, we are interested in performance and better result then the sequential program. The number of generations is fixed and is equal to NG for each active process. This is different of traditional treatment in which data are divided by processor number and each data slice is processed by one processor. The symbol ESOW-n denotes the Pool of computers utilised in this test, where n is the number of processors. In order to realise this experiment, we used four computers Pentium-II 266 MHz and three Pentium III- 650 MHz, running Windows and Linux. The parallel program was done using ESOW with 2, 3, 4, 5, 6 and 7 computers.

TABLE VII
Results with f1(x)

Method	N.G.	f1(x)	Std Dev	time (ms)
Yao & Liu	750	2.5×10^{-4}	6.8×10^{-5}	-
Lopes&Zair	750	2.32×10^{-11}	1.7×10^{-11}	125632.13
Lopes&Zair	300	42.91×10^{-4}	17.2×10^{-4}	43028.78
ESOW – 2 *	300	10.08×10^{-4}	2.33×10^{-5}	58880.14
ESOW – 3 *	300	5.54×10^{-5}	2.55×10^{-5}	67120.56
ESOW – 4 *	300	1.86×10^{-5}	5.96×10^{-6}	81595.23
ESOW – 5 *	300	1.75×10^{-6}	7.99×10^{-7}	85870.02
ESOW – 6 *	300	1.095×10^{-8}	5.01×10^{-9}	94420.45
ESOW – 7 *	300	2.25×10^{-8}	2.15×10^{-8}	100015.36

* the test was done with 40 migrations of parents
N.G. – Number of Generation

The values in table VII and table VIII show that the parallel algorithm running on ESOW is faster than the sequential solution. The parallel solutions is compatible or better than the sequential solution.

TABLE VIII
Results with f2(x)

Method	N.G.	f1(x)	Std Dev	time (ms)
Yao & Liu	2000	9.7×10^{-4}	4.2×10^{-4}	-
Lopes&Zair	2000	3.07485×10^{-4}	4×10^{-19}	96247.00
Lopes&Zair	500	3.085×10^{-4}	7×10^{-7}	14224.15
ESOW – 2 *	500	3.07486×10^{-4}	5.54×10^{-11}	34856.83
ESOW – 3 *	500	3.07486×10^{-4}	5.60×10^{-11}	27930.72
ESOW – 4 *	500	3.07485×10^{-4}	3.93×10^{-11}	31665.00
ESOW – 5 *	500	3.07485×10^{-4}	4.87×10^{-15}	45480.03
ESOW – 6 *	500	3.07485×10^{-4}	7.27×10^{-17}	45860.36
ESOW – 7 *	500	3.07485×10^{-4}	2.03×10^{-19}	46905.42

* the test was done with 50 migrations of parents
N.G. – number of generation

It is important to say that migration of parents denotes the transference of information among active objects (objects which has the process with evolution algorithm), that is, the best parent of each active object is transported to others actives objects in determined number of generations using passive objects (List_soc and Stack_soc).

These tests show the power of object-oriented in distributed/parallel programming, in special our environment.

V. Comparison with others Environments

Just as Charlotte, ESOW provides distributed shared memory without relying on operating system or compiler support. Both are implemented using only Java, this provide the same level of security, heterogeneity, and portability of Java. Charlotte and ESOW are unlike in

schematic of load balancing. Charlotte use eager scheduling to provide load balancing and tolerance to fault, this implies that worker processes running on slow machines ask for jobs less frequency, this result in load balancing, while ESOW needs know some parameters to do load balancing with migration. ESOW is reliable because it has replication and persistence.

Javalin and ESOW utilise an intermediary component for process communication between clients and hosts. Clients in Javalin are responsible to do load balancing and tolerance to fault while ESOW utilise its Kernel. Because this, ESOW is more transparent than Javalin. The communication in Javalin is done to mimic the native Java UDP classes in java.net, while ESOW uses Java RMI.

Ninflet and ESOW are developed using RMI as mechanisms of communication. Both systems can be used not only to construct a global computing environment, but also could be used for implementing a virtual parallel computer on a cluster of workstations. Ninflet requires explicit checkpointing by Ninflet programmer and allow RMI connection to hosts other than the one where applet was loaded. This is bad, because allow arbitrary intrusion of other RMI applications. While ESOW has same security police of applets.

VI. Conclusion

This paper presents an environment for distributed and parallel programming called ESOW. It is designed to allow programmers and developers of applications on the Internet and Intranet doing distributed/parallel programs without knowing details about network, schedule of task, load balancing and fault tolerance. The ESOW potential is on using networks of computers as a pool of processors.

With our proposed environment, it is possible to take advantage of the computational resources of idle computers on Internet or Intranet. The most of the tied computers the internet does not have half of its used potential.

ESOW is a parallel virtual machine which allow computers share process and data, utilising load balancing, tolerance to fault and transform Internet in one Pool of processors.

References

[ABD 00] ABDELOUAHAB, Zair; LOPES, Denivaldo. Development of an Environment for Supporting Parallel and Distributed Application with Java. Proceedings of the Software Engineering and Applications (SEA 2000), IASTED, Las Vegas, USA. November 2000, p. 343-348.

[CHO 98] CHOI, Chang Ho et al. CSMonitor: A visual Client/Server Monitor for Corba-based Distributed Applications. Proceedings of the Asia-Pacific Software Engeneering Conference (APSEC'98), Taipei, Taiwan, Dec. 1-4, 1998, p. 338-345.

[KEL 99] KELEHER, Peter. Decentralized Replicated-Object Protocols. In the 18th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC '99), April 1999.

[MAS 99] MASSEN, Jason, et al. An Efficient Implementation of Java's Remote Method Invocation. PpoPP'99, ACM Symposium on Principles and Practice of Parallel Programming, Atlanta, GA. May 1999.

[TAN 99] TANENBAUM, Andrew S. et al. From Remote to Physically Distributed Objects. Proc. 7th IEEE Workshop on Future Trends of Distributed Computing Systems, Cape Town, South Africa, December 1999, pp. 47-52.

[THI 00] THITIKAMOL, Kritchalach; KELEHER, Peter. Thread Migration and Load Balancing in Non-Dedicated Environments. In The 2000 International Parallel and Distributed Processing Symposium, May 2000.

[OMG 99] OMG. *The Commom Object Resquest Broker: Architecture and Specification*. Object Management. Minor revision 2.3.1. October, 1999.

[NIE 99] NIEUWPOORT, Rob, et al. Wide-Area Parallel Computing in Java. To appear in JavaGrande 99, ACM 1999 Java Grande Conference, Palo Alto, California, USA. June 1999.

[RAP 00] RAPTIS, Konstantinos, SPINELLIS, Diomidis and KATSIKAS, Sokratis. Java as Distributed Object Glue. In World Computer Congress 2000. Beijing, China. August, 2000.

[STE 99] STETS, Robert J. Leveraging Symetric Multiprocessors and System Area Networks in Software Distributed Shared Memory. Thesis of Doctor of Philosophy. University of Rochester, New York. 1999.

[WAT 95] WATTS, Jerrel. A Practical Approach to Dynamic Load Balancing. California Institute of Tecnhology. Dissertation of Master Science. 1995.

[RAC 97] RACKL, Günther. Load Distribution for Corba Environments. Technische Universität München. Diplomarbeit. 1997.

[TAK 98] TAKAGI, H. et al. Ninflet: a Migratabel Parallel Objects Framework using Java. In ACM 1998 workshop on Java for High-Performance Network Computing, 1998.

[CHR 97] CHRISTIANSEN, O. and CAPELLO, P. Javalin: Internet-Based Parallel Computing Using Java. Concurrency: Practice and Experience, Vol. 9(11): 1139-1160, November 1997.

[BAR 96] BARATLOO, Arash, KARAUL, M. Charlotte: Metacomputing on the Web. Proc. of the 9 th International Conference on Parallel and Distributed Computing Systems, 1996.

[YAO 99] YAO, Xin, LIU, Yong, LIN, Guangming . Evolution Programming Made Faster. IEEE transactions on Evolutionary Computing, 3(2):82-102, July 1999.