Tiger : Towards Object-Oriented Distributed and Parallel Programming in Global Environment

Youn-Hee Han, Chan Yeol Park, Chong-Sun Hwang, and Young-Sik Jeoung*

Department of Computer Science and Engineering, Korea University,
* Division of Computer and Communication Engineering, WonKwang University.

Abstract. Numerous attempts have been made at providing a Internet-wide global computing infrastructure. Unfortunately, none of them presents programming constructs related with object distribution, dispatching, migration and concurrency, which provide maximum portability and high transparency to a programmer. This paper proposes a World Wide Web based global computing infrastructure called *Tiger* which supports numerous participating machines and active objects. Tiger provides noble object-oriented programming constructs supporting object distribution, dispatching, migration and concurrency. Tiger infrastructure and the given programming constructs allow a programmer to easily develop a well composing object-oriented distributed and parallel application using globally extended resources. To show the speedup achieved by Tiger, a parallel genetic-neuro-fuzzy algorithm requiring much computational time is experimented.

1 Introduction

Utilization of resources available to a network of workstations has well served to gain enough computing resources in order to execute a computationally intensive application[1–3]. In these case, however, a significant administration is required to run the application. Programs are written in traditional languages like Fortran or C. The download of codes is done manually by the administrator that is also responsible to install and configure system. Therefore, the size of participating hosts group is small and power for parallel and distributed computation may be restricted.

Recently, the World-Wide-Web (hereinafter referred as the Web) has became the largest virtual system. There have been substantial changes in this Internet age, as such result of proliferation of low priced powerful hosts connected by high-speed links. At any given moment, however, the many are idle. An appealing idea is to utilize these hosts for running applications that require large computational power. Such noble computing paradigm has been called *Global Computing*[4, 5].

Some of the obstacles common to global computing are the heterogeneity of the participating hosts, difficulties in administering distributed applications, and security concerns of users. The Java language and Java applets with Java-capable

Web browsers have successfully addressed some of these problems. Platform-independent portability is genuinely supported by the Java execution environment. The growing number of Java-capable browsers able to load applets from remote side reduces administration difficulties. The browsers execute untrusted applets in a trusted environment, which alleviates some of the users' security concerns. Also, Java is a simple, robust, multithreaded language and is designed to support applications on networks. Therefore, Java and Java applets with Java-capable Web browsers have become a good candidate for constructing global computing platform[6, 7].

In the current state of the art, however, developing a distributed and parallel application using global resources requires specialist knowledge beyond that needed to develop an application run on a single machine[8, 9]. In a global system, many machines participate and numerous objects come in and come out. Therefore, it must allow the incremental growth of a system without the user's awareness. Besides, it should achieve a consistent and predictable performance level regardless of the change of the system structure or load distribution. In object-oriented paradigm, therefore, it is required to dispatch an existing object from local machine to a another machine and move the object to a different machine from the dispatched machine. Both CORBA and JavaRMI, two good solutions constructing distributed applications, do not provide any solution about those related by global computing. Nevertheless, they put a heavy burden on programmers since they require many modifications of existing sequential code. Our first challenge is to deal with requisites for object-oriented global systems, - scalability, load balancing using object dispatching and object migration - that are not issues in a classical sequential or multithreaded application executed within one machine.

In Java, threads are only a mechanism to express concurrency on a single host but do not allow to express the concurrency between remote hosts. Therefore, currently, a huge gap exists between multithreaded and distributed application. Our second challenge is to provide the noble global distributed and parallel programming paradigm constructs which can combine existing sequential programming constructs into several global application related constructs, that is, distribution, dispatching, migration and concurrency. Global parallel and distributed programming can be simplified by implementing the constructs into the classical Java language. They provide the programmer with maximum portability and high transparency since they allow him or her to write the programs in a shared-memory style.

This paper is organized as follows: a description of our system and object model are given in section 2. This is followed by a detailed discussion about distributed and parallel programming using global resources in section 3. Experimental results are presented in section 4. Finally, conclusions are presented in section 5.

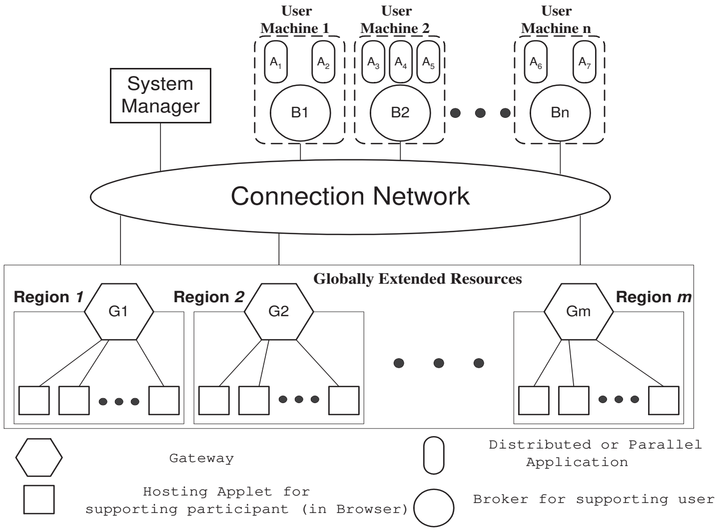

Fig. 1. Tiger System Architecture

2 Tiger System and Object Models

2.1 System Architecture

We propose a design of a object-oriented global computing infrastructure called *Tiger*. Our system model consists of six kinds of major components: users, brokers, hosting applets, gateways, regions, a manager(Fig. 1).

- *Users* wish to use extra computing power in order to run large distributed and parallel application.
- *Brokers* manage user application in the same node and coordinate all communication between the applications and Tiger. It is necessary for a user to execute a broker before executing his or her application.
- *Hosting applets* allow their CPU resources to be used by other users. The general form of them is an Java applet in a Java-enabled web browser.
- *Gateways* manage parts of available hosting applets and coordinate all communication between them and other components in Tiger system. We assume that every gateway serves exactly one
- *Regions* consist of a gateway and hosting applets managed by the gateway. Regions are generated by grouping hosting applets which show similar round-trip time of communication for a gateway managing them.
- *A manager* registers and manages participating brokers and gateways.

Also, it is responsible for following main activities: location management for all the dispatched and mobile objects in Tiger, load distribution among all participating gateways. Whereas, the primary function of gateways consists of followings: location management for objects visiting its associated region, load distribution among hosting applets managed in the region, There are two important reasons why our model must provide gateways. The first reason is generated in order to distribute the manager's massive load. Because of the heavy network traffic generated by the many brokers and hosts, the manager may become a bottleneck. To reduce the traffic to the manager, one natural solution is to distribute the manager functions in several other things. The second reason comes from severe limitations on the capabilities of Java applets, since the applets are used to execute untrusted programs from the Internet. Applets cannot create server socket to accept any incoming connection and Java-capable browsers disallow applets from establishing a network connection except to the machine where they were loaded from. Using gateways as the intermediate message-exchange nodes, we allow an applet to communicate with any applets in the same region(Fig. 2(a)) or in different regions(Fig. 2(b)). Gateways use these communication route mainly for object migration in order to balance loads among hosting applets.

Both a broker and a gateway intermediate message exchanges between a user application and a hosting applet(Fig. 2(c)). Two different user applications can communicate through each associated broker(Fig. 2(d)). These communication route is mainly used for remote method call.

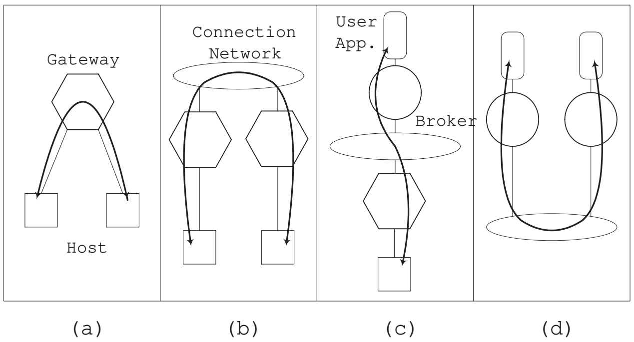

Fig. 2. The Communication Model

2.2 Triple-Object Model

Tiger is designed as an object-oriented global system which expresses computation as a set of autonomous communicating entities, *objects*. An object is a compound entity consisting of four components: *an identifier*, *a class(or type)* describing member fields and methods, *properties*, and *a location address*. The components are separated into the variant part and the invariant part. The variant part of a object includes properties and a location address. Properties are stored in an encapsulated object and may be updated after executing a defined method. Location address is, by default, allocated as the address of machine where the object is created, and updated whenever the object moves between hosts. Also, There are an identifier and a class in invariant part. The identifier is globally unique and is used to make references to the object. The class contains Java byte-code defining the object's member fields and methods. It can be maintained discretely and stored in files or archives on a local system or on a network server. A object's identifier and class components remain invariant across all possible modifications of the object's properties and location address.

We use a triple-object model in Tiger. There are *normal(local) objects*, *distributed objects* and *remote mobile objects*. Normal objects are the same those as standard Java objects. Distributed objects are placed to other user machines rather than the one which local objects are placed in. Like a distributed object in CORBA or JavaRMI, they are accessible remotely and locations of them are fixed to the user machine where they are created. Mobile objects are similar to distributed objects except that they can change its current location from a user machine/ hosting applet to other hosting applets. Distributed objects and Mobile object are the basic units for distribution and concurrency. Users have to specify explicitly which objects are distributed or migrated.

A requested method call on a remote object is either executed or queued depending upon an object's state at the time of arrival. An object can be one of three state: *dormant*, *active*, and *waiting*. The object in dormant is not currently executing any method, and there are no one in message queue. The object in active state is currently executing a method. The object in waiting state is waiting for a specific response to a method call issued while in the active state. When an acceptable response arrives, the object will return to the active state.

3 Distributed and Parallel Programming

When designing an object-oriented application, programmers start with high-level abstractions and turn them into objects and classes. Programmers are usually eager in modeling and algorithmic issues about the application. Deciding how to distribute or move some objects into several machines is definitely a lower-level issue. Our programming model concentrates on a clear separation between high-level design and lower-level implementation issues such as object distribution, object migration, and controlling concurrent activities. When dealing with globally distributed objects, the power of Tiger lies in that any client processes or threads in local machine can directly interact with a server object that live on a globally distributed machine, a user machine or hosting applet through remote method call although the server object migrates among them.

3.1 TigerObject Interfaces

A server object must declare its service via an interface. It does this by extending *TigerObject interface*. Each remote method is declared in the interface. Like JavaRMI, client stub and server skeletons are generated from this interface and the implementation of client and server object is done using this interface.

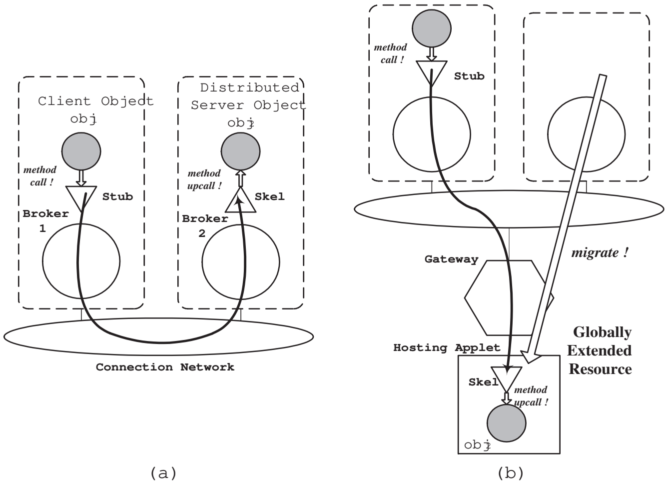

Fig. 3. The Object Distributing Mechanisms

3.2 Objects-distributing Mechanisms

Tiger provides us with the two application programming interfaces(hereinafter referred as the APIs) related with distributing server objects.

- `turnDistributedObj(TigerObject obj, String name)`
  This converts a existing local object `obj` into a distributed object at any time after its creation. The distributed object becomes accessible remotely and its location is fixed to the server machine indicated by `name`. The stub object is registered to naming subsystem together with a given name. The stub acts as a handle for client to reference the remote server object. The client can call methods on the stub, which are routed by the client broker to the server broker, where the skeleton executes the method upcall on the actual server object(Fig. 3(a)). This mechanism is similar to that in the CORBA and JavaRMI.

- `turnMobileObj(TigerObject obj, String name)`
- `turnMobileObj(TigerObject obj, String name, Region destRegion)`
  Unlike CORBA and JavaRMI, on the other hand, Tiger allows the local or distributed server object, `obj` to migrate from the server machine to a hosting applet in globally extended resource. It is noted that a client process or thread do not know a detailed location of migrated hosting applet. They do not present any such a information or present only region information, `destRegion`. That is, Tiger provides users with migration transparency. Also, the location of migrated server object is not fixed to the first hosting applet and the server object can re-migrate to another hosting applet without awareness of a user. Through such migration mechanism, we can reduce the burden on server machine and gain the performance benefits(Fig. 3(b)).

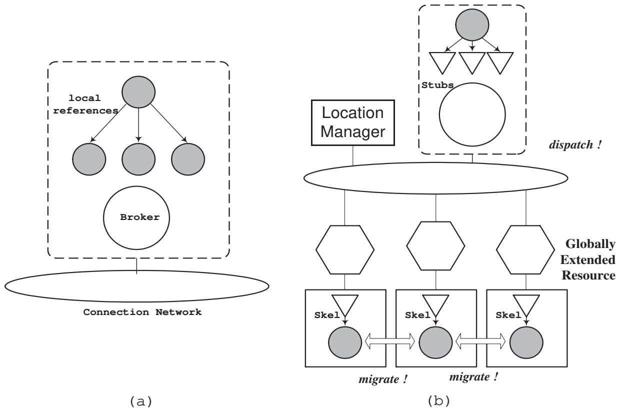

Fig. 4. The Parallel Object Dispatch

3.3 Parallel Objects-dispatching Mechanisms

Object-oriented parallel programs are largely divided into two parts: one being *the main control program*, which provides a whole body for solving a given problem; the other being *TigerObject*, which describes tasks to be executed in parallel. The main control program must dispatch a number of TigerObjects into the globally extended resource and call remote methods on the dispatched objects. Besides, it can watch loads imposed on each region, can balance those using object migration and can retract the dispatched objects from the globally extended resource to local host.

Tiger provides us with the one API related with dispatching parallel TigerObjects.

- `turnMobileObj(TigerObject obj)`
- `turnMobileObj(TigerObject obj, Region destRegion)`
  This overloaded method dispatches a existing local object `obj` into a hosting applet in the globally extended resource. The dispatched object becomes accessible remotely from the dispatching user machine(Fig. 4). It is not required to provide the name of object since the object is only used by the main control program. Tiger itself, inside interior, allocates the globally unique identifier to the object. It is noted that its location is not fixed to the dispatched hosting applet and the dispatched object can re-migrate to another hosting applet.

3.4 Remote Method Call and Concurrency

Concurrency can be used to perform given applications in parallel on several machine or on one machine controlling different threads, only simulating parallelism. Although method calls of Java are only synchronous, there are three constructs provided to call a remote method in Tiger: *synchronous*, *asynchronous*, *one-way*.

A synchronous remote method call is typically used in order to distribute local objects so that users can concurrently access them that reside on different nodes. The calling object, however, must blocked until the result is returned. By default, Tiger executes calls to remote objects in a synchronous fashion.

Asynchronous remote method call has a *future* interaction style, in which the caller may proceed until the result is needed. At the time, the caller is blocked until the result becomes available. If the result has been supplied, the caller resumes and continues. For supporting a asynchronous fashion, we provide `AsyncReply` and `AsyncReplySet` classes and `asyncMethodCall(TigerObject obj, String methodName, Object[] args)` API. By using these classes, it is possible to issue an asynchronous call to a remote object that is executed in parallel.

The third construct, one-way remote method call, is also asynchronous. The caller, however, will not retain any thing related by this method call, and the callee will never have to reply to it. It has *fire-and-forget* style.

To distinguish among these three constructs, each remote method, which is declared in a user-defined interface extending TigerObject interface, must throw one among *SyncMethodCallException*, *AsyncMethodCallException*, and *onewayMethodCallException*. These three exceptions specify what the method's behavior style is.

4 Experimental Results

We conducted experiments to show the speedup achieved by Tiger. The target application for experiments is a parallel genetic-neuro-fuzzy algorithm. Genetic-neuro-fuzzy algorithms are a hybrid method for neuro-fuzzy systems based on genetic algorithms, in order to find the global solution for the parameters of neuro-fuzzy system. They always begin by generating an initial population randomly, after they encode the parameter into chromosomes. Then, they run iteratively repeating the following processes until they arrive at a predetermined ending conditions: *extracting fuzzy rules, self-tuning, fitness evaluation, reproduction, performing genetic operators(crossover and mutation).* It requires much computational time to construct a fuzzy system from a chromosome. The communication time, however, does not affect the total processing time. So, Tiger is suitable to execute genetic-neuro-fuzzy algortithms

A major characteristic of our genetic-neuro-fuzzy algorithm is that the capability-based adaptive load balancing is supported to reduce total working time for obtaining optimal fuzzy system. Let T_i be the time that is taken to execute the operations of chromosmes allocated to Region i and NF_i be the sum of the number of fuzzy rules processed in each chromosome allocated to the Region i. Then, the number of chromosomes which will be allocated to Region i at next generation, N_i, is defined by

$$N_i = N_c \cdot \frac{C_i}{\sum_{i=0}^{N_f-1} C_i} \ , \quad where \ C_i = \frac{NF_i}{T_i} \tag{1}$$

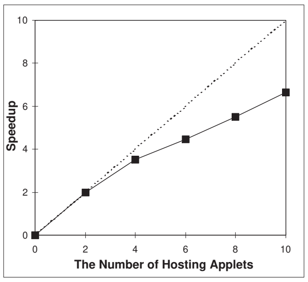

Fig. 5. Speedup for the number of hosting applets

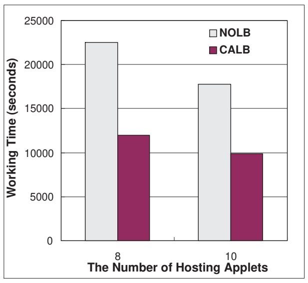

Fig. 6. The efficiency of load balancing

In equation (1), N_c is the total number of chromosomes given in the system, N_f is the number of Regions participated in the system currently, and C_i is the capability of Regions i based the number of fuzzy rules processed in unit time.

Using equation (1), we can decide the number of chromosomes which are allocated to each Region at next generation and can move some chromosomes to other Regions using Tiger's APIs. The goal of our algorithm is to make fuzzy system which can approximate the three input nonlinear fuction defined by

$$output = (1 + x^{0.5} + y^{-1} + z^{-1.5})^2 \tag{2}$$

A total of 216 training data are sampled uniformly from input ranges $[1.6] \times [1.6] \times [1.6]$. The parameters for the genetic-neuro-fuzzy algorithms used in the experiment summarized in Table 1.

population size	50	number of generation	50	chromosome size	48
prob. of crossover	0.3	prob. of mutation	0.15	learning iteration	50

Table 1. The parameters for the genetic-neuro-fuzzy algorithms

The system manager runs on Pentium 333Mhz and three gateways run on Pentium 200, 233, 266Mhz using Java virtual machine in JDK 1.2.1. Ten hosting applets run in the Netscape Communicator 4.5 or Internet Explorer 4.0 on heterogeneous machines connected 10Mb/s Ethernet. Figure 5 presents the speedup curve according to the change of the number of hosting applets. In the figure, as the number of hosting applets increases, the speedup is more enhanced and speedup curve become more remote from linear line.

Figure 6 shows the efficiency of capability-based adaptive load balancing scheme. In the figure, *NOLB* represents scheme that does not use a load balancing and *CALB* represents that use a capability-based load balancing. We conducted the experiment on eight hosting applets and ten hosting applets, respectively. When the number of hosting applets is eight *CALB* shows 1.89 times as performance as *NOLB* shows. Similarly, when the number of hosting applets is ten *CALB* shows 1.80 times as performance as *NOLB* shows.

5 Conclusions

We have designed and implemented Tiger, a global computing infrastructure able to use the computing resources of numerous machine connected in Web. Tiger provides noble object-oriented programming constructs supporting object distribution, dispatching, migration and concurrency. These constructs, together with Tiger infrastructure, allow a programmer to develop easily a well composing object-oriented distributed and parallel application using globally extended resources.

We are currently working on an extended version of Tiger which supports a mechanism with fault tolerance, global load sharing and stealing, result verification and user privacy. We believe that the future version of Tiger will become a robust and high-performance global computing infrastructure.

References

1. T.E. Anderson, D.E. Culler, and D.A. Paterson, A case for NOW(Network of Workstations)., *IEEE Micro*, vol.15 no.1, pp.54-64, February 1995.
2. N.J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J.N. Seizovic, and W.S. Myrinet, A gigabit-per-second local area network, *IEEE Micro*, vol.15 no.1, pp.29-36, February 1995.
3. T.M. Warschko, J.M. Blum, and W.F. Tichy, The ParaStation project: using workstations as building blocks for parallel computing, *Proc. Intl. Conf. on PDPTA'96*, Sunnyvale, CA, pp.375-386, August 1996.
4. J.E. Baldeshwieler, R.D. Blumofe, and E.A. Brewer, ATLAS : An infrastructure for global computing, *Proc. of the 7th ACM SIGOPS european workshop on system support for world wide applications*, 1996.
5. T. Brecht, H. Sandhu, M. Shan, and J. Talbot, ParaWeb: Towards world-wide supercomputing, *Proc. of the 7th ACM SIGOPS European Workshop*, on system support for world wide applications, pp. 181-188, September 1996.
6. K.M. Chandy, B. Dimitrov, H. Le, J. Mandleson, M. Richardson, A. Rifkin, P.A.G. Sivilotti, W. Tanaka, and L. Weisman, A world-wide distributed system using Java and the Internet, *Proc. of the 5th IEEE Intl. symposium on high performance distributed computing*, Syracuse, NY, August 1996.
7. A. Baratloo, M. Karaul, H. Karl, and Z.M. Kedem, An infrastructure for network computing with Java applets, *ACM Workshop on Java for High-Performance Network Computing*, Palo Alto, California, February, 1998
8. D. Caromel, W. Klauser, J. Vayssiere, Towards seamless computing and metacomputing in Java, *Proc. of concurrency practice and experience*, pp. 1043-1061, September 1998.
9. M. Boger, F. Wienberg, W. Lamersdorf, Dejay: Unifying concurrency and distribution to achieve a distributed Java, *TOOLS99*, Nancy, France, June, 1999.