SCHEDAE INFORMATICAE
MCCLIX VOLUME 11 2002

The Object Oriented Platform for the Process Migration in the Heterogeneous Networks

Piotr Uhruski, Zdzislaw Onderka
Institute of Computer Science, Jagiellonian University,
Nawojki 11, 30-072 Krakow, Poland
e-mail: uhruski, onderka@ii.uj.edu.pl

Abstract. This paper describes the object oriented approach to the design of a task migration platform in a heterogeneous computer network. The *load sharing* and *load balancing* problems are discussed. The *load sharing* problem consisting of three parts: an information policy, a location policy and a transfer policy was presented [3]. The migration Software Development Kit (SDK) for an application, which should meet the well defined requirements, is defined. The above mentioned SDK was applied to the example of multiplying the given vector by the given matrix, which is a frequent subproblem in the CAE calculations.

Keywords: Process migration, heterogeneous network, distributed application, object oriented design and programming, Java, CORBA, load sharing, load balancing.

1. Introduction

The networks of the heterogeneous workstations become a standard computing environment for calculations of the *Grand Challenge Problems* [14]. For example, it could be any CAE (*Computer Aided Engineering*) application for the large scale problem [16, 15]. The real time of the sequential execution of such computation is very long and it is often enlarged by the frequent disk transmitions when the RAM memory is not large enough to store all the computation data. Such problem occurs when the computer RAM is

too small at all, but also when other applications, requiring a huge memory for their own calculations, are executed at the same time on the same machine. Moreover, the users require sometimes the peek of a computing power on a single workstation. The computing power of the computer network is under-utilized most of the time [1, 2]. A consequence of the above mentioned features is utilization of the dynamic *load sharing* mechanisms i.e. if there are computers with a very low load and free resources (like a big amount of the RAM memory), they can be used to relieve over-utilized workstations. It could be achieved by the *preemptive process migration* [3, 4]. In such migration the execution of the task is suspended on the current machine and then resumed on the other one [4].

In this paper the object oriented (OO) approach to the *preemptive process migration* is presented. The migrated process is an object, whose execution could be stopped, serialized on the current machine and then deserialized on the other one and resumed there. Additionally, the OO techniques enable to define the base classes, which can be used to define the inherited classes of objects for the user-specific calculations. These base classes are created in the form of the *Software Development Kit* defined for the presented migration platform.

2. Process migration facility

Processes migration has been implemented in various systems to gain more computing power from a workstations network understand as a virtual machine. The paper [5] summarizes these implementations and gives a good background to processes migration problems. The mentioned paper divides all migration efforts into the following groups:

- Processes migration over UNIX, which bases on an operating system acting as a common interface for any hardware. The Freedman and Condor systems are mentioned here as examples.

- Different process migration platforms, working both in homogeneous (Charlotte, Accent, RHODOS) and heterogeneous (Amoeba, Locus, MOSIX, Sprite) networks. The presented systems were created as experimental systems to investigate such issues as a transparent access, failure notification, resources protection, load distribution strategies and fault tolerance.

- Migration facilities in systems based on the micro−kernel paradigm, such as Mach and Chorus. Because of the kernel low complexity, the process must incorporate many data within itself, what makes it easier to migrate.

- Systems providing object migration are the last group. In addition to the systems presented in the paper (e.g., Olsen, Chorus/COOL, Emerald, SOS) it is worth to mention new systems utilizing the mobile agents paradigm. The object migration focuses on abstract entities – objects, with different levels of granularity. Such migration problems include defining (and leading an object to) a persistent state before migration and migrating complex objects like linked lists.

The mentioned paper [5] presents also very extensive bibliography for all mentioned systems.

3. Migration object model

The presented migration platform design is focused on easy-defined migratable tasks that should simplify the platform usage and allow users to incorporate migratable tasks in their applications. We have assumed that many task functions are common for any task that can be created by the user, so a base class for a task has been proposed. This base class implements the main properties of the tasks that are described in the following sections, so the custom task developer can focus only on task-specific details.

Because of a task easy-definition requirement we have decided to use an object oriented programming techniques. Every custom task class is inherited from the task base class, so each task interacts in the same way with its environment (via the task base class), only internal information processing is different. Actually, we have created a task framework where the custom task internal processing is encapsulated by the base class.

On the other side, a simple task definition results in task class implementation complexity. The task uses high-level calls, which must be translated into low-level machine calls. For example, consider network communication, with OSI/ISO layers [11]. Usage of the communication protocol from the OSI/ISO application layer is easier than using it from the OSI/ISO network layer, but network layer calls have only to go through 3 OSI/ISO layers to reach the destination host. The same efficiency decrease is natural for all object oriented applications.

To develop the OO migration platform, Java programming language and CORBA communication framework have been used.

3.1. Java Programming Language

Among many languages, Java gives us the possibility to use the same, compiled code, on any computers in a heterogeneous network. Java classes

are compiled into machine independent *byte-code*, which is interpreted by a machine and operating system dependent Java Virtual Machine (JVM) [12].

3.2. CORBA framework

To migrate tasks in a heterogeneous network, we need to provide a transparent network communication protocol for computation servers. The second requirement is the protocol specification that gives servers the mechanism to find each others in the network. These two requirements are satisfied by the CORBA framework, as described in [13].

In addition, using the CORBA Naming Service, the task executing on a given host can be exposed to any application from a different machine in the network. This feature enables the developer to create a migratable or non-migratable (dedicated to a certain server machine architecture) task performing specialized computations that resides on one machine and receives requests from foreign applications. The described feature can improve the presented migration platform, but it is not yet implemented.

4. Migration algorithm

The migration algorithm must be based on a well-defined load sharing policy. As defined in [1] the load sharing algorithm is based on three main parts: an information policy, a transfer policy and a location policy. The information policy specifies, which information is used while deciding about the process migration and how this information is available in the system. The transfer policy defines the need to migrate for each process and the location policy defines target a migration host for each task.

4.1. The information policy

In our work, we wanted not only to monitor the current state of the machine loads, but also to get the certain machines hardware parameters and to calculate the machine load forecast. Therefore we have decided to create a separate system described in [9]. Its main functions are:

- Monitoring the current machines load in the network.
- Predicting the machines load in different time horizons.

- Collecting information about machine various hardware parameters including the free and total amount of the machine RAM memory, free and total amount of the total machine memory (including virtual memory) and the machine type (e.g., parallel, vector).

The separate agent system mentioned above is available through the CORBA services on the examined machines, so it can be accessible to any foreign application.

Using previously listed information provided by that agent system, a migration platform can be widely used in distributed CAE tasks, which require to be well-fitted to available machines. As described in [6, 15], the CAE mesh generator can create many small tasks using given information about the available machines RAM memory. Finally we get well-fitted tasks that can be allocated and computed without exceeding machines hardware limitations (RAM). Such a solution is possible thanks to the presented problem fine granularity. Smaller tasks can also be distributed more efficiently in the network.

4.2. The transfer policy

As defined in [1], the request of the task to migrate is defined by the transfer policy. The presented platform provides three decision ways. The first is a *migration-on-demand*, which is the simplest one. We gave the possibility to migrate any running task to a specified machine. The migration server administrator is able to select a task from its user interface, enter a destination host address and then request selected tasks migration. We assume, that a destination host cannot refuse received tasks, but it might not resume them immediately. It can forward new tasks to any other hosts that it is connected with. Therefore, this transfer policy can be used to solve any potential task migration deadlocks, resulting from *task-decided* (see below) migration with unreachable requirements.

The server implements the second *server-decided* migration algorithm. This transfer policy is similar to the one described in [1]. It includes two load thresholds, T_{low} and T_{high}, that are used by the server internal migration decision algorithm. This algorithm shall work as a local server load guard. If the current server load is high or a high load is predicted, the algorithm decides that the server is overloaded (system resources are fully used) and requests migration of the local tasks to another computation servers. On the other hand, the migration server uses the decision algorithm, if it is asked by another server to receive new tasks. If the current load is low or a low load is predicted in the future, a new task can be accepted and resumed locally. Here is the decision algorithm skeleton sketch for the current machine load case:

1. The current machine load $< T_{low}$. The server is available. It accepts tasks.

2. T_{low} <= the current machine load $< T_{high}$. The server does not accept tasks and it does not request migration too. It is in normal load state.

3. T_{high} <= the current machine load. The server requests migration, it is overloaded.

We assume that T_{high} must be greater than $T_{low} + 1$ to avoid a boomerang effect [1]. Each task may implement the *task-decided* transfer policy. Such task has the access to the machine load information service through its execution environment on the server. It might decide that it needs to be migrated and sends request to the server to initialize migration. The task can specify its requirements for the destination machine, which include the maximum machine load and free memory.

The *task-decided* policy can be widely used in CAE tasks, where the developer knows the requirements for the created task (memory and CPU time requirements). For example the task inverting matrix might be started on any server and wait for matrix data. When it receives the data and the current machine free memory prohibits inversion, the agent requests to be migrated into a less memory-loaded machine.

The *task-decided* transfer policy may be implemented by every agent, but it uses interfaces exposed by a tasks common base class, which links with the server environment interfaces to provide an access to the server machine load system. This is a good example how OO design simplifies an agent structure and emphasizes an agent encapsulation without loosing agent universality.

4.3. The location policy

When the decision about what should be migrated is taken, we need to decide where to migrate it. This step might be used to improve load balancing at any time in the network. Each server computer must be connected manually by the administrator with all servers that it will migrate tasks to (sink servers). This configuration step can be done by the administrator using a server user interface or by a server configuration script, which is read at the server startup. The list of sink servers can be modified by the administrator at the server runtime, so the administrator can dynamically adjust it.

The process of task migration must be designed to avoid any potential deadlocks (specified migration requirements result in lack of available destination hosts) and omit migration to a server, which load has changed during single task migration. Therefore, migration is represented as a logical transaction between two server machines. See [10] for more details on transactions topics. In the designed platform, when migration is requested, a new object

representing a migration transaction is created. It receives the task to migrate and the list of migration sink servers. Migration is done using a *two-phase commit protocol*, further described in [10]. Migration transaction stages are organized as follows:

1. All potential migration sinks are asked, whether they can accept the migrated task with its requirements.

2. From all sinks, one that matches all requirements is selected. If more than one sink matches all requirements, the one with the smallest machine load is selected by the transaction supervisor object. If there is no machine that matches the request and migration was *server-decided*, the server reports this to the administrator, who decides whether the task must be suspended (the machine load system provides a load prognosis) or migrated manually, as described in Section 4.2.

   If migration is *task-decided*, the task is responsible for the decision whether migration shall proceed or not. First of all, the task can lower its destination machine requirements and request migration again. This gives the ability to dynamically adjust the task behaviour to migration network load status. The task may check a machine load forecast, send message to its owner or even end its execution. The task can also report its state to the administrator, who decides about the task execution as in the previous *server-decided* case.

3. Selected migration sink is asked once again for migration. This acknowledgment locks a sink host so it cannot accept migration from any other hosts until this certain migration has been finished. This guarantees that the migrated agent will execute in the environment that it asked for.

4. The task is serialized on the source host and moved to the sink host. All messages from the tasks in-coming queue are moved to the sink host.

5. The sink host resumes the task and notifies the source host when the task is successfully resumed. The transaction object notifies the task owner (see Section 5.2) about a new task location.

6. The transaction object is de-allocated and all migration resources are freed.

5. Migration platform

5.1. Task specification

5.1.1. Task execution

Every task implements some logic that the user wants to execute. It might be computation, messages receiving/posting or any other activity. Task expects two execution "ways" from its environment.

- The task expects from its environment that it will be executed in a separate thread, to perform it's internal computation. This kind of a task activity can be called an *active loop*. For example, the simple CAE task body may implement multiplication of the given matrix by the given vector.
- Each task, after performing some computations in the *active loop*, might want to send a message to its creator or wait for a message from anyone. The task environment shall contain two message queues: incoming and outgoing. Message posting is an asynchronous operation, it does not influence a task execution thread, but the message receiving process is available in two forms. The task, while it is executed in its own thread can suspend itself (blocking message check) and wait for any new messages, or it can wait actively, asking for any new message (non-blocking message check). The message queues mechanism allows to developer to create a new kind of task that works as service for any potential client application. For example, it can be a task multiplying an array by a vector, which receives computation input data from a client. It waits suspended for the new data and when it receives it, the computation is performed. After results are returned (as a message from this task), the task suspends its execution again, waiting for a new data. This feature can be joined with CORBA services and as a result we got a distributed network of easy-accessed services.

These two execution ways can be implemented using a *thread paradigm*. Operating system defines a thread as a part of a machine process that executes within the same memory area and processor time as the whole process. One process can contain many threads and each of them receives part of process execution time. In many languages as C++ or Java, a thread body is defined as a loop. If this loop exits, the thread exits. First implementation of our system lets the user to create single threaded tasks. Such a task can be serialized and send to another server. Further implementations may also consider multi-threaded agents. Remember, that every thread is represented by a single object, so we are still serializing single objects. Only a migration server must be modified to launch many threads for one agent.

5.1.2. Task data

We assume that each task requires some data to work with. In our implementation, this data must be incorporated into a task object as an attribute, so it will be serialized with the task. One important Java feature is that serialization operates not only on a single object, but also on an object's graph. A task body can be divided into separate objects, performing different works and containing different data. This allows the task developer to design a task better than using only one object. Single objects can implement task execution stages with their own data structures.

Further improvements will introduce distributed data descriptors, which might work as distributed handles to the task data. This will allow the task to migrate without moving large portions of data.

5.1.3. Task environment

Each task is executed in similar the environment, provided by the migration server that executes the task. To make a task run or stop, that task has to implement some well known interface, that the environment can use to operate on every task. As we have defined in previous chapters, the presented migration platform provides the task framework that the developer uses to create a custom task. The framework consists of a task base class, that implements interface used by the execution environment.

5.1.4. Task migration details

In the migration solution presented here, the task class is implemented using Java language. Java provides a special interface *Runnable* that defines an interface of a thread object. It contains only one method, run(), which is used to define the task execution body. The migration sever treats tasks as any other threads defined using Java language.

- **Task Serialization**

  To perform the task migration, it must be serialized into some data stream (string object here) and then it can be migrated. Serialization support in Java language allows a whole object serialization, with all attributes (if these attributes types are serializable too). To make a class serializable, it must only implement the Java *Serializable* interface. Of course Java serialization supports objects deserialization too.

- **Task Pause/Resume**

  The task might not always be ready to pause, therefore the task developer has to define points in the task body where the task is ready to be stopped and migrated. The execution environment asks the task to stop, and when the task execution reaches the migration-ready point, then the task is serialized and migrated. The task body can check the current system environment and if there is not enough resources to proceed with task execution, the task might request migration. The task resume process requires then one more task element, an *execution stage*. The following example shows how the *execution stage* works. Lets consider a task with the following internal structure:

  (1) Wait for data >
  (2) Perfom computation >
  (3) Send Results >
  (4) End

  The task can migrate after the end of each stage. But if the task were migrated after it finishes stage **(1)**, it must be resumed not from the beginning, but from the begining of phase **(2)**. Therefore at the end of stage **(1)**, the task stores internal information that stage **(2)** is the current one and checks if it was requested to stop.

  The task can also define its migration-ready points within stage processing. If the presented task stage **(2)** would multiply the received array and vector, it can migrate after each matrix row is multiplied by the vector. The task must store multiplying loop indices internally as attributes and after task resume, start computation from the stored indices values.

5.2. Task SDK

As it was presented above, OO techniques enabled us to separate task internal logic (which is different for all tasks classes) from basic task operations. The Migration *Software Development Kit* (SDK) shall contain a task base class: *TaskBase*, which provides all base functionalities that every task shall contain including the task internal structure and external task environment interfaces. This class will be recognized by the migration server and used to perform all task operations. SDK shall also give the user an easy access to the migration servers. This is accomplished by a *TaskController* class, which serves as a gateway for all agents. To send a task to the server, the user application must complete the following steps:

1. Create an object of a *TaskController* type, giving a migration host address as constructor parameter. The task controller links itself with the

specified host by obtaining reference to the CORBA migration service registered on the host.

2. Create a new instance of a task, prepare it to execution and pass it to the task controller. The task controller assigns that task a global unique ID (composed of a machine unique ID string from Java and a machine IP number) and forwards the task to the migration server where it is executed.

3. When the task is finished, the task controller receives task results, and passes it to the user code. Results can be received in the form of a message from the task or as the task object with internal result data.

Additionally, task SDK is also responsible for defining the task execution context, in which a task is executed on the server. This execution context includes in-coming and out-going message queues, a machine load system interface and interface of the server migration manager allowing the task to request *task-decided* migration.

5.2.1. Sample task

The Appendix A presents the code of a sample task multiplying the given vector by the given matrix. The introduced skeleton code presents the message queues and execution stages implemented in the platform. The feature ofthe message queue that blocks a caller thread execution until a new message arrives is used by the task to wait for computation data − the matrix and vector.

The sample task class depicts how the task computation is divided into two main stages: an initialization and main computing loop. When a task object is created by the platform server, first its **initComputing()** routine is called, so the task can initialize internal data structures. This routine is called only once in task life, by the first platform server that enacts the task execution.

The task main computations are embedded in the **run()** method. This routine is called each time the task is resumed after migration (or hibernation due to the resources lack), thus it requires the previously presented execution stages − see Section 5.1.4.

6. Migration server

The server works as the execution environment of migratable agents. It does not need to know any task internal details, since it uses *TaskBase* in-

terfaces as defined in SDK. The server can be decomposed into three main parts:

- **Tasks Executor**
  This part manages all task threads. It is responsible for serializing/deserializing tasks and running them. Each task is wrapped into a container, which is a *TaskHolder* object. The holder provides the message queues and the task environment interfaces. It is used by the migration transaction object to hold incoming messages, which are forwarded to a new task server after the migration is successful.
- **Machine Load Monitor System**
  This subsystem works as a wrapper for the load monitoring system described in [9]. The server subsystem exposes load-monitoring interfaces to both the server migration manager (which performs server-decided migration) and tasks, which perform *task-decided* migration.
- **Migration Manager**
  It provides migration transaction objects and the list of available hosts, see Section 4.2 for details.

7. Experiences

The migration platform presented above has been implemented using Java JDK1.3 SE, with Java ORB included with this software bundle. It has been tested on machines working on Linux RedHat 7.0 and Windows 2000. Two types of agents were implemented:

7.1. Array by vector multiplication

The task was similar to the agent presented in the "Task SDK" section. After execution it waits for array and vector messages. When both data structures are received, the agent checks the current machine load state and depending on it migration is requested or not. After computation is done, the task is destroyed.

7.2. SBS PCG algorithm

This implementation is based on the SBS PCG implementation presented in [8]. A new application, playing the role of the SBS master process, was

created in Java. It uses the Task SDK to create the slave tasks objects and send them to the computation server. The SBS slave process is a migratable task created using the Task SDK.

7.3. Experiences summary

Two presented living applications that were created using the platform let us make some general observations regarding application effectiveness and maintainability. First, the SBS PCG algorithm implemented with the platform works about 4–5 times slower than C language. This time estimations were done in the testing environment with those two applications running sequentially with a rough total computation time estimation. This putsthe platform based solution far behind the native application. On the other side, it took only few days to implement the SBC PCG algorithm using the platform SDK.

As can be seen from the sample task code in the previous chapters, the platform provides all required communication, transaction and migration mechanisms that have to be used by each application. This makes the platform based solution far more flexible and maintainable than the native solution, which is created once and can not be easily scaled up.

The platform decreases single task execution time, but in total, it speeds up a whole application development process. What's more, thanks to it different scheduling politics, an application can be scaled up without rewriting core parts. We have here the living sample of a "write once – use many times" OO programming law.

8. Improvements and future works

This section summarizes all improvements that are mentioned in the article.

- The tasks themselves should be available as CORBA services – Sections 3.2 and 5.1.1.
- The task data shall be separated from the task body, introduce distributed data descriptors – Section 5.1.2.
- Allow the developer to create migratable multi-threaded tasks – Section 5.1.1.

Additionally, the presented task will be in future used as a base platform for a CAD system, partially described in [6, 15, 16].

9. References

[1] Bernard G., Simatic M.; A Decentralized and Efficient Algorithm for Load Sharing in Network Workstations, EurOpen '91 Conference Materials, Tromso, Norway 1991.

[2] Theimer M.M., Lantz K.A.; Finding Idle Machines in a Workstation-Based Distributed System, *IEEE Trans. on Software Engineering*, Vol 15(11), Nov 1989.

[3] Bernard G., Steve D., Simatic M.; A Survey of Load Sharing in Networks of Workstations, *Distributed Systems Engineering*, pp. 75–86, 1993.

[4] Alard E., Bernard G.; Preemptive Process Migration in Networks of Unix Workstations, *Proceedings of the 7th International Symposium on Computer Science and Information Sciences*, Antalaya, Turkey 1992.

[5] Nuttall M.; A brief survey of systems providing process or object migration facilities, *Operating Systems Review*, Vol.(24), pp. 64−80, 1994.

[6] Schaefer R., Toporkiewicz W., Grochowski M.; Meshless Partitioning for Parallel PDE Solution, *Proceedings of the Third International Conference On Parallel Processing and Mathematics*, Kazimierz Dolny, Poland, 1999.

[7] Myśliwiec G., Sipowicz J.; Optimal Management of a Distributed Linear Solver, *Proc. of XIII Intern. Conf. on Computer Methods in Mechanics*, Poznań, May 1997.

[8] Myśliwiec G., Sipowicz J.; *Distributed iterative SBS-type linear solvers*, Master Thesis, Jagiellonian University, Institute of Computer Science, Kraków 1997.

[9] Lepiarz M., Onderka Z.; Agent System for Load Monitoring of The Heterogenous Computer Network, Accepted to PPAM'2001 conference, Nałęczów, Poland, 2001.

[10] Tanenbaum A.S.; *Distributed Operating Systems*, Prentice-Hall International, 1995.

[11] Stevens W.R.; *Unix Network Programming*, Prentice-Hall Inc., 1990.

[12] Eckel B.; *Thinking In Java. Second Edition*, Prentice-Hall Inc., 2000.

[13] Siegel J.; *CORBA 3: Fundamentals And Programming, Second Edition*, OMG Press, 2000.

[14] Agarwal R.K.; Parallel Computers and Large Problems in Industry, *Proc. Computational Methods in Applied Sciences*, Elsevier Science Publisher, 1992.

[15] Onderka Z.; *Stochastic Control of the Distributed Scalable Applications. Application in the CAE Technology*, Ph.D. Thesis, Technical University AGH, Department of Computer Science, Kraków, Poland, 1997.

[16] Onderka Z.; Schaefer R.; Markov Chain Based Management of Large Scale Distributed Computations of Earthen Dam Leakages, *Lecture Notes in Computer Science*, No 1215, pp. 49–64, Springer-Verlag, 1997.

Received May 14, 2002

10. Appendix A

The code of a sample task class is presented. It waits for a matrix and a vector and then multiplies them.

```
import Migration.SDK.*;

public class ArrayMulVectorTask extends TaskBase {
//
// Internal task data structures
private double [][] arrayValues;
private double [] vectorValues;
private double [] resultValues;
private int mulRowIndex;
private int mulColIndex;

public ArrayMulVectorTask(double [][] array, double [] vector) {
    //
    // Allocate internal data structures and copy
    // constructor arguments to internal attributes
    ...
}
//
// TaskBase overloads
public void initComputing() {
    this.mulRowIndex = 0;
    this.mulColIndex = 0;
    super.initComputing();
}
//
// Task body
public void run() {
```

```
    switch (this.runStage) {
    case 1:
        //
        // First we wait for messages.
        // These are blocking calls
        TaskMsg msg = super.getTaskExecutionContext().
            getInMsgQueue().
            getMessageBlock();
        this.arrayValues = decodeArrayFromMsg(msg);
        //
        // The same with vector values
        ...
        this.markStage(2); break;
        case 2:
        //
        // Compute
        while ( this.mulRowIndex ¡ this.arrayValues.length)
        {
            this.resultValues[ this.mulRowIndex ] = 0;
            this.mulColIndex = 0;
            while ( this.mulColIndex ¡ his.arrayValues[0].length) {
                this.resultValues[this.mulRowIndex]+=
                    this.arrayValues[this.mulRowIndex] [this.mulColIndex]*
                    this.vectorValues[this.mulColIndex];
                this.mulColIndex ++;
                if (pauseRequested() == true) {
                    return;
                }
            }
            this.mulRowIndex ++;
            if (pauseRequested() == true) {
                return;
            }
        }
        this.markTaskAsFinished();
    }
}
```