JEM: a different approach of heterogeneous parallel and distributed computing based on Java[***]

Serge Chaumette and Asier Ugarte

LaBRI, Laboratoire Bordelais de Recherche en Informatique,
Université Bordeaux I, FRANCE

What you are reading is not the final version of this paper. We have been too late for the dead-line and this is why we submit it as is. We will send a new version Tuesday June 15th.

Sorry for the trouble
Serge Chaumette

Abstract. JEM is a distributed platform based on Java-RMI[1]/CORBA[6] that makes it possible to access heterogeneous resources distributed over a network in a seamless manner. It provides the same interface to all resources integrated inside the system.
In this paper we present two of the main features of the JEM platform. We first illustrate the usage of the desktop-like interface which we have developped on top of the kernel of the platform. This makes it possible for instance to drag a file over a printer to have it printed without knowing where it is located. Second, we describe the entity of execution wich we have integrated inside the platform to handle fine-grain parallelism. These entities are some sort of mix between mobile agents and remotely accessible threads. The possibility to make fine-grain parallelism provided by our platform is one of the main features that makes it different from existing systems. Indeed, existing systems provide a sort of worldwide virtual computer better suited for large-grain parallelism.

1 Introduction

Many research activity is being carried out around what is referred to as Meta, Web or Grid computing. The aim of the most significant projects is to provide the scientific computing community with a seamless virtual computer that would

---

[*] Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. The author is independent of Sun Microsystems, Inc.

[**] This work is partly supported by the Université Bordeaux I, the Region Aquitaine and the CNRS.

span the whole world. Such a system must provide a simple to use desktop-like user interface that makes it possible to use the environment in a point and click way.

The platform which we are developping tries to produce such a platform points, still providing efficient fine-grain parallelism, which is often not in the scope of other platforms. First, it provides a Jini-like "plug-and-participate" structure [?]. This part of the platform will not be detailed here and the user is referred to [3] for more information. On top of this infrastructure we have developped a graphical desktop which is both a proof of concept and a mandatory tool if we want final users to use the system in a seamless manner. Second, it offers all the necessary infrastructure for a meta-computing platform. At the same time, it also provides the required mechanism that will make it possible to use fine-grain parallelism efficiently. In this paper we focus on the two main features of the Jem platform: first, its desktop like interface to network resources based on a volonteer plug and participate mechanism. Second, we present the unit of execution of the platform, Activities for mobility and Tasks for remotly controlable Threads. The platform is based on Java-RMi and currently also run on top of CORBA.

The rest of this paper is organized as follows. In section 2 we present projects wich are in some manner related to what we are doing in the Jem platform. Section 3 shows the most significant features that make it possible to understand how the Jem platform works. We then concentrate on two of the major views an external user of the platfon can have : a desktop distributed environment in the one hand (section 4), and a possibly fine-grain parallel computing architecture (section 5) int the other hand. We eventually conclude in section 6 and sketch future directions of the project and usages of the Jem platform.

2 Related work

In this section we present the main features of those systems which our system best compares to. These are Millenium, Jini, Legion and Globus. Many more systems exist, and the user is referred to [2] for more more information.

Millenium[7] is a Microsoft research project. The aim of this project is to provide a distributed system to "eliminate the distinction between distributed and local computing". To achieve this goal it provides "agressive abstraction" that hides any physical information from the programmer. The system is self configuring: a new hardware equipment can join the system or can be removed in case of failure. Of course scalability, security and resource management are strong concerns in the project.

Two prototypes have been developped, Coign and Continuum, to illustrate the operations of Millenium on COM components. Work is also being carried out on communication, both to make it more efficient and to make it customizable.

Jini is a software architecture developped by Sun. Its aim is to provide a distributed "plug-and-participate" architecture. It makes it possible to share resources, to virtualize the location of resources, to manage easily a set of resources. These resources can be physical, like printers for instance, or logical like pieces of software.

To join a Jini community, a component is plugged physically into the system. It joins the community by registering its service by a look up service. This service is now available for any one (other service) willing to use it. To do so, a software component will querry the lookup service. It will then receive an interface to communicate with the service. There is then a direct communication using RMI with the service juste located.

Globus[4] is a project, the aim of which is to provide the infrastructure to set-up a computational grid. These efforts come to life with the Glous Ubiquitous Supercomputing testbed which is the interconnection of computers using the technologies and software component provided within the Globus "bag of services". The main basic principles can be summarized as an "hourglass" and translucent interfaces. The principle of the hourglass is to provide a single simple interface to different available low-level services, interface that can be used to build new high level, distributed services. For instance, the communication layer of Globus is the Nexus library. It can at the same time be implemented on top of several available low-level communication libraries (IP, shared memory, ..) and it can be used to develop higher level libraries such as MPI. This provides for portability of applications developped using Nexus. Nevertheless it might be worth controling the way Nexus is mapped to the underlyinug protocol for instance to gain some efficiency when possible because of a specific application. This is what is refered to as translucent interfaces. Soem attributes cab be used to set-up parameters that control the mapping of nexus to underlying layers.

Legion[5] is a project wich is carried out at the University of Virginia. The aim of this project is to build a "Worldwilde Virtual Computer". Legion is an object-based opened system. It provides scheduling, fault tolerance, site autonomy and many security possiblities. One of the main aims of the system is to provide users with parallelism that uses the underlying possibly many computers available worldwide. In legion everything is an object. This architecture closely ressembles the CORBA architecture. Objects in some sens publish methods that can be invoked by other objects. The naming mecanism of Legion closely ressembles that of CORBA. Although Legion provides supports parallel libraries such as MPI, and also supports wrapping of parallel components, it is not appropriate for fine-grain parallelism.

3 Structure of the Jem paltform

Jem provides an homogeneous access to resources of an heterogeneous network of resources. These resources can be either physical – printers, screens, processors – or logical – some software code, ... –. This part of Jem ressembles the Jini environment. Resources are wrapped inside some oriented glue that present all of them with the same interface to the system. Each resource can be fed using a `copyIn` method and results can be obtained using a `copyOut` call. We will not enter the details of the internal structure of the kernel in this paper. For furthet details the reader is refered to [3] that describes the kernel of Jem in details.

In the rest of this paper, we focus on the exteral view of the system. We present the desktop interface that was built on top of the kernel. We also present the

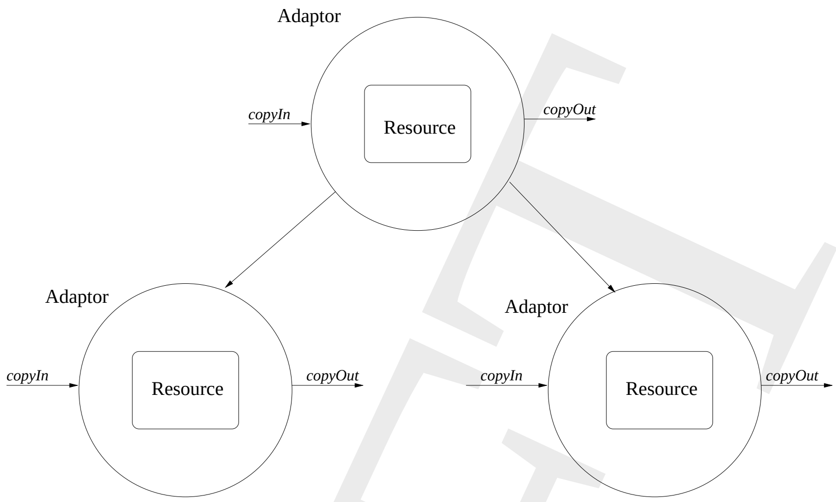


Fig. 1.

notions that make it possible to develop real computational systems inside the platform.

4 JEM as a desktop platform

A desktop like interface has been developped. The aim of this "point and click" interface is to provide users with an environment which is straightforward to use. The fact that the components of the system are possibly distributed over a network is hidden from the final user. The user can easily manage all types of components simply using drag and drop. Figures 2 and 3 show this interface.

For instance, the user can drag a file shown on the desktop to a printer. This causes the file to be printed. The user does not need to care about the physical location of these objects in the network. This hability to migrate resources is made possible by the underlying platform.

5 JEM as a parallel computing platform

JEM is also a distributed and parallel environment. The unit of concurrency used is defined as the Activity.

An *Activity* is some sort of mobile agent. It can move (migrate) from host to host inside the Jem Platform and continue its activity after migration. Activities are implemented keeping the Jem philosophy in mind : it is a resource

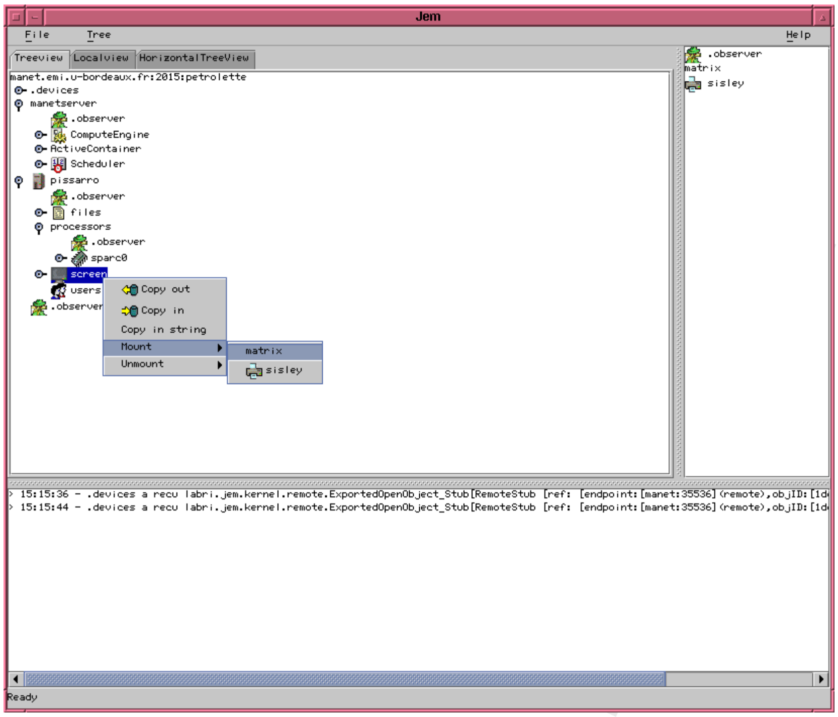


Fig. 2. The user desktop interface build on top of JEM

managed through the set of generic operations defined in the previous section. It will handled like a screen or a processor, respects the notions of parenthood and hierarchy of our platform.

Due to the fact that we we do not want to control the compilation, we are sometimes unnable to prevent the activities from doing operations which we have difficulties to handle. An activity is free to access any physical local resource, which are inaccessible or inconsistant in a remote way ; think of a file descriptor in a file system for instance. This has two main consequences : an activity accessing a local physical resource cannot migrate. It must stay stationnary while the resource is accessed. Also we cannot make the activity to migrate. It is to the activity to decide when to migrate.

Note that a thread of execution, that it is a *java.lang.Thread*, is in some sense a physical resource. In fact, one of the things that keep us from controling totally the *Activity* is the Threads use. Of course, an *Activity* implementation is free to create any threads and use them at its own convenience. In return, the Activity becomes stationnary.

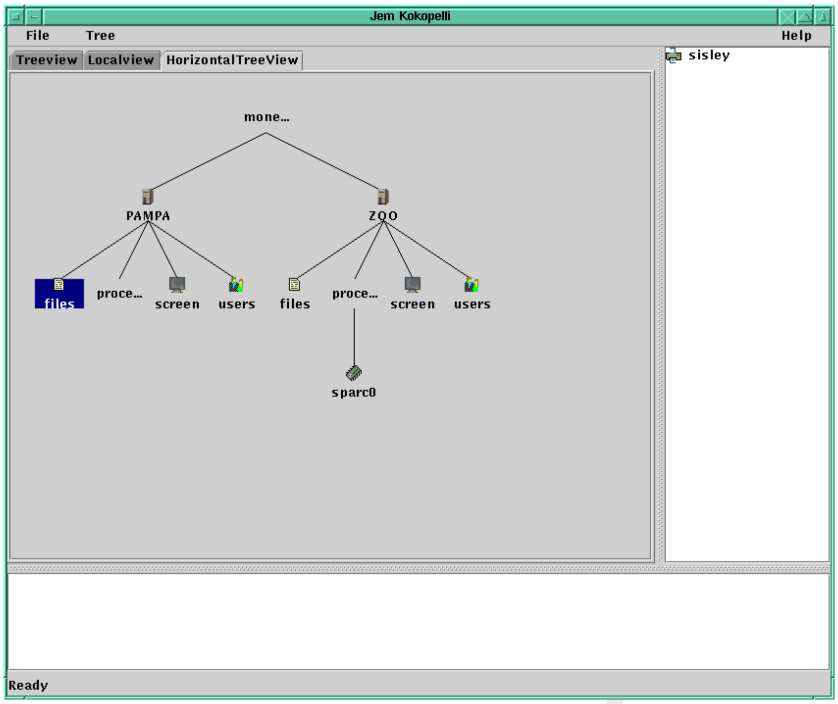


Fig. 3. The horizontal view mode

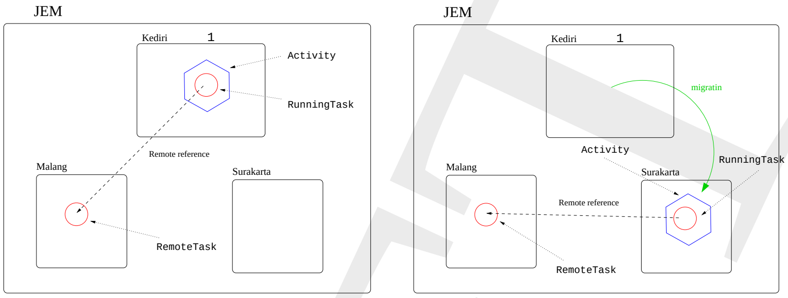
5.1 Introducing the Tasks

We have seen that an activity cannot access physical resources still staying in a stable state that makes migration possible. To solve this problem we introduce a new unit of execution which we call a **Task**.

A **Task** can be seen as a java thread the execution of which may be remote. The main difference between an **Activity** and a **Task** is that a **Task** cannot migrate. The **task** remains local to the server where it was started. Nevertheless, a **Task** may be remote to the **Activity** that created it. Furthermore, a *Task* can create other remote **Tasks** in a recursive manner.

In a more generic way we talk of asynchronous Tasks. The Task remains active for an unspecified amount of time, running in a remote *Server*, and the *Activity* retrieves the result from the *task* in a asynchronous way. Note that the *Asynchronous Task* reminds the thread concept, but in a more powerfull way: we have the concurrency of the threads, but we can specify the locality of the execution and the *Jem Acitvity* remains in a stable state, so that it can migrate.

5.2 How does it all work ?



(a) (b) Migration of an Antivity

In what follows, for the sake of explanation, we use the names of machines for servers. Figure 4(a) shows an activity running inside the server Kediri. This activity remotely creates a task inside a server called Malang. The activity can communicate with the task through the RunningTask handle that was returned by the system at starting time. The activity then migrates to the Surakarta server, as shown figures, carrying with it the RunningTask handle which allows to communicate with the remote task.

Activities From a programmer's point of view the process of creating an activity consists in extending the *Activity* class shown programm 1.

— Program 1 : prgs/Activity.txt —

```
public class Activity implements java.io.Serializable,
                                 java.lang.Runnable{
  [...]
  public final void regule(Location _location)...
  public void run(){}
  [...]
}
```

— Program 1 : prgs/Activity.txt —

The *Activity* class shoud not be directly instantiated, but extended to create *Activities* inside the Jem Platform ; this class obeys the same usage pattern as `java.lang.Thread` in the Java API. The newly created activity will be able to migrate by invoking the *regule* method. This method can only be called by the activity on itself : it is up to the activity to decide when to migrate. The reason for this is that the activity is the only one to know when it is in a stable state that allows migration. Furthermore the policy to use to chose the destination of the migration is left to the activity. These criteria can be the most powerfull server available, the server with the lowest load, . . .

Program 2 : prgs/MyActivity.txt

```
public class MonActivity extends Activity{
  public void run(){
  [...]
  regule(Location Policy);
  [...]
  }
}
```

Program 2 : prgs/MyActivity.txt

Tasks An activity willing to create its own threads of execution, still willing to keep the capacity to migrate cannot use a `java.lang.Thread` : it will use the *Task* class that we provide instead. Task is both a Runnable and a Serializable, i.e. it implements these interfaces. To create a Jem Task the programmer extends it and redefines its run method. When starting a task, the Jem kernel returns a handle to a *RunningTask*. A RunningTask handle allows a user to interact with a remote Task, thus behaving like a proxy. An activity can migrate with a RunningTask, and continue to interact with the remote Task by way of the RunningTask handle. The RunningTask is created by the Jem Kernel when starting a Task, and returned to the caller of the start method. The Task and the RunningTask classes should not be directly instantiated.

Program 3 : prgs/runningtask.txt

```
public final class RunningTask implements java.io.Serializable{

    [...]
    public String getName()...
    public int getPriority()...
    public void setPriority(int _priority)...
    public void join()...
    [...]
}
```

Program 3 : prgs/runningtask.txt

Note that a Task, is like an Activity, a jem resource, and it respects all the principles defined previously. Task make it possible for an activity to access a local physical resource that otherwise will prevent the activity to migrate. In fact, the *RunningTask* will allow us to accomplish all the services usually found in a thread implementation : synchronizations, management of threads, scheduling… Figures 4(a) and 4(b) illustrate the activity and task mechanisms.

Program 4 : prgs/MyApplication.txt

```
public MyTask extends Task{
  public void run(){
    [...]
  }
}

public MyActivity(){
  public void run(){
     [...]
     MyTask _task = new MyTask();
     RunningTask _runningTask = start(_task);
     [...]
     regule(Location Policy);
     [...]
     _runningTask.join();
     [...]
  }
}

public class MyApplication implements Application{
  public void main(Context context, String args[])
         throws labri.jem.kernel.remote.TransportException{
    MyActivity _activity = new MyActivity();
    [...]
  }
}
```

Program 4 : prgs/MyApplication.txt

5.3 Task scheduling

Activities and tasks provide the developers with two powerfull and flexible mechanisms to implement distributed and parallel applications on the Jem platform. Nevertheless, the Jem platform is a shared platform, thus the tasks of different users might compete inside the same Java virtual machine. Therefore, we provide our own configurable scheduler. At present, a priority-timesliced policy scheduler is available. Figure 4 illustrates operation of the current scheduler implementation. It uses priority levels, and then round robin time sliced scheduling at a given priority level.

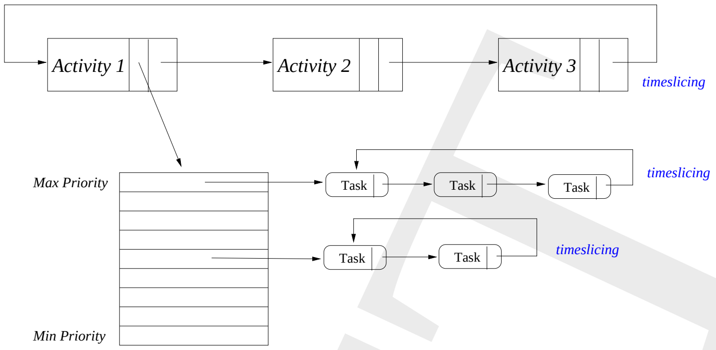


Fig. 4. The current scheduler policy

6 Conclusion and future work

Jem is still in the phase of development. We keep on improving the functionnalities, still getting better insight of what the kernel must be like, and how to make it even more opened so as to keep our development adaptable. We are concious that some mandatory features are still missing. Among these are access rights, encryption of communication, ... We believe that the new laws regarding the usage of encryption in France will make it easier for us to deal with these problems using public-key/private-key technics, credentials and access control lists. The implementation will partly take place inside he kernel of the system and as Jem objects in charge of the security. The development of the desktop was a student project. The desktop is built on top of the kernel functionnalities, and the result has been quite successfull.

The advantage of the Jem platform is that it can both provide large-grain and fine-grain paralllism. The next phase regarding this point is double. First we intend to use this feature to develop applications that would exploit our scheduling possibilities using fine-grain paralllism. Second, we intend to take advantage of the migration possibilities of our platform even further. We are on the way to integrate the PM2 library in the Jem Platform. PM2 is a ibrary that makes it possible to develop C threads that can migrate. Even though migrating in C is of course more limited than migrating in java – where heterogeneous target platforms can be considered – we will most likely gain efficiency and be able to take legacy code into account.

As of writting, we are now in the process of considering our platform for deployment inside our Laboratory for system adminstration, and as a tool used within the framework of the parallel and distributed lecture of the Compute Science Engineer School which is an associated school of University Bordeaux 1.

References

1. K. Arnold and J. Gosling. *The Java programming language.* Addison-Wesley, 1996.

2. S. Chaumette. JEM experimentation environMent for Java compared to other objet systems. Technical report, LaBRI, 1998. To be published.
3. Serge Chaumette. JEM-DOOS: the Java based Distributed Objects Operating System of the JEM project. In Denis Caromel and al., editors, *Proceedings of the Second International Symposium on Object Oriented Parallel Environments, ISCOPE 98, Santa Fe, NM, USA*, volume 1505, pages 135–142, december 1998.
4. Ian Foster and Kesselman Car. The globus project: A status report. In *Proceedings of IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.
5. A.S. Grimshaw, A. Wulf, and the Legion team. Legion. The next logical step toward the world-wide virtual computer. *Communications of the ACM*, 40(1), 1997.
6. J. Siegel. *CORBA, Fundamental and Programming*. Wiley, 1996.
7. Robert P. Fitzgerald Christopher W. Fraser Michael B. Jones Todd B. Knoblock Rick Rashid William J. Bolosky, Richard P. Draves. Operating system directions for the next millenium.