JPDC: Java Package for Distributed Computing

Umberto Ferraro Petrillo, Delfina Malandrino, Alberto Negro
Nadia Romano, Vittorio Scarano
Dipartimento di Informatica ed Applicazioni "R.M. Capocelli"
Università di Salerno, 84081, Baronissi (SA) – Italy
E-mail: {umbfer,delmal,alberto,vitsca}@dia.unisa.it

Abstract

In this paper we present Java Package for Distributed Computing ($\mathcal{JPDC}$), a toolkit for implementing and testing distributed algorithms in Java. $\mathcal{JPDC}$'s goals are to simplify the development of distributed algorithms by defining an high-level programming interface. The interface is very close to the pseudo-code formalism commonly used to describe algorithms and allows, at the same time, the implementation and deployment in a truly distributed setting. Moreover, $\mathcal{JPDC}$ also provides a friendly interface that can be used both to visualize the algorithm behavior and both to interact with it. This is especially useful for teaching environments where complex algorithms can be much better debugged, understood and validated by implementing and running them.

1 Introduction

The explosive growth of Internet and the availability of inexpensive hardware are promoting an increasing interest in Distributed Computing because of its characteristics of scalability, efficient resources sharing, fault tolerance and reliability. Unfortunately the development, the testing and the deployment of a distributed application may be a difficult task due to the complexity of commonly used distributed programming environments. This problem heavily affects the real diffusion of distributed applications both for teaching and scientific purposes. As a matter of fact, frameworks for distributed programming such as MPI or CORBA provide only a low level framework for implementing distributed algorithms. They require good programming skills and force the programmer to take care of many technical details.

We believe an ideal framework to design and deploy a distributed application should meet (among others) the following requirements. First, it should implement a programming interface very close to pseudo-code often used for describing and presenting distributed algorithms. Second, the framework should allow to run implemented algorithms in a truly distributed environment. The framework should also be flexible enough to allow the user to change on-the-fly some parameters such as the delay on each communication link (to simulate the congestion on the network) and the fault probability of a node or of a link. Third, it should be possible to visualize at different level of details the algorithms' behavior in order to improve its comprehension. By using such a framework, a user should be able not only to easily implement an algorithm but also to experiment its behavior. For example, it should be possible to test an algorithm by providing it a particular input, examining the various messages that are exchanged during the execution and, finally, observing the changes in the state of every node of the system by means of a graphical representation.

$\mathcal{JPDC}$ is a toolkit for helping the design and the implementation of distributed algorithms. It requires a limited effort to the programmer since it hides the details of the communication between the nodes of the graph. As a result, distributed algorithms can be easily coded since the programmer needs to implement a standard set of $\mathcal{JPDC}$ methods. Such methods define the behavior of each node that participates to a distributed computation by using standard high-level APIs featuring send/receive communication primitives (the communication model is Message Passing). Therefore, coding an algorithm by $\mathcal{JPDC}$ becomes a straightforward task since there is an immediate mapping between pseudo-code and Java code.

Organization of the paper: $\mathcal{JPDC}$ structure is outlined, first, by showing, in Section 2, how $\mathcal{JPDC}$ can be used to code a distributed algorithm. The example shown offers the chance to smoothly introduce some of $\mathcal{JPDC}$ data structures and APIs to allow an easy coding of the algorithm. Then, we briefly describe $\mathcal{JPDC}$ architecture and implementation (in Section 3) and its graphical user interface (in Section 4) with few details on its profiling component. Finally, in Section 5 we conclude by describing the state of the art and $\mathcal{JPDC}$'s role in the field.

2 Coding distributed algorithms

While designing $\mathcal{JPDC}$ one of our main goals has been to simplify the implementation of distributed algorithms.

Pseudo-Code	Implementation

For each node Pi (with 0<i<=n-1)
 	
Init:
parent is nil
children and other are the empty set
	public class FloodingSTNode extends Node {
	
	private Identifier parent;
	private NodeList children;
	private NodeList other;
	
	public FloodingSTNode(String graphname,Integer pid)
	throws Exception{
	super(graphname,pid.intValue());
	parent = null;
	children = new NodeList();
	other = new NodeList();
	}
	}

Figure 1: Flooding algorithm *initialization*.

To this end, we wished to offer to the end-users a programming interface very close to the pseudo-code used to present and describe distributed algorithms. We used the pseudo-code as presented by Attiya and Welch in their well-known textbook of distributed algorithm [1].

In this way, users are able to write code which closely follows the original algorithms' specifications while not caring about implementation details. Such a result has been made possible by properly exploiting the capabilities of Object Oriented programming and by developing a standard Application Programming Interface (API) implementing the typical algorithmic patterns found in distributed algorithms. A typical $\mathcal{JPDC}$ usage session can be described in two phases. In the first phase, the user implements a distributed algorithm using the $\mathcal{JPDC}$ APIs. In the second phase (see Section 3), the algorithm is run providing as input the definition of the weighted graph describing the network topology to be used.

Implementing an algorithm with $\mathcal{JPDC}$ is almost as simple as writing its pseudo-code. The developer needs to write a class that is derived by $\mathcal{JPDC}$'s class `Node` and fill-in-the-blanks to specify the behavior in four phases of the algorithm, namely: *initialization*, *execution* (start-up or wake-up), *message reception handling* and *termination*. Instances of this class represent each node participating to the algorithm.

To clarify how to use $\mathcal{JPDC}$ and also to provide a mild introduction to its APIs and its structure, we present here the implementation of the standard flooding algorithm.

Flooding Algorithm. Given a network of connected nodes, the idea behind the flooding algorithm is to recursively propagate messages starting from a fixed root node in order to reach all the remaining nodes. In our case, we will consider the variant for building a spanning tree of the underlying network. At initialization time, we choose a node that will be the root of the tree being grown. This node will send to all its neighbors a "BeParent" message announcing itself as their parent. Upon receiving a "BeParent" message, a node has two possibilities: if this message was never received before then the sender is assumed as its parent otherwise the message is acknowledged by replying with a "Reject" message. At the end of the computation the spanning tree will cover all nodes reachable from the root node.

When implementing a distributed algorithm in $\mathcal{JPDC}$ the Java class to be created must derive the `Node` abstract class. The `FloodingSTNode` object, as presented in Figure 1, implements the flooding algorithm we have described above. It defines three variables: `parent`, used to store the reference to the parent node, `children` and `other`, used to partition the list of neighbor nodes. These information are maintained using a `NodeList`, a $\mathcal{JPDC}$ data structure that can be used for easily handling group of nodes.

In Figure 1 we also reported the initialization steps of the algorithm, implemented in the constructor of the `FloodingSTNode` object. We can point out here that the constructor of each class implementing an algorithm in $\mathcal{JPDC}$ takes as arguments the name of the current "session" and a unique ID number. These information are needed for building the distributed computation network and are automatically supplied by the underlying $\mathcal{JPDC}$ architecture. Moreover, in the same class we need to provide also a numeric coding for messages types.

At start-up all the nodes wait for incoming messages. After being awoken, node with ID "r" starts to "flood" its neighbors by a message of type "BeParent" and acts as the root of the spanning tree. Sending a message in $\mathcal{JPDC}$ is accomplished in two steps: first, the sender needs to prepare the message object[1] by indicating its ID and the message type. Then, the sender uses one of the many $\mathcal{JPDC}$ APIs available to send the message. These APIs offer the user to send messages either to a single `Node` or to a `NodeList` therefore allowing a versatile use by selecting a subset of nodes as destination. Both pseudo-code and implementation are presented in Figure 2.

[1]We provide a simple `Message` class that encloses message type and sender ID. One can easily subclass `Message` and enclose additional data in the object.

Pseudo-Code	Implementation

send BEPARENT to all neighbors
 	
parent:=r
	public void execute() throws RemoteException {
	Message msg;
	msg = new Message(getPid(), FloodingMsgType.BEPARENT);
	sendAll(msg);
	parent = getPid();
	}

Figure 2: Flooding algorithm *execution* (start-up) as issued by node with ID 'r'.

When a node receives a "BeParent" it checks if it has already a parent, if so then the message is acknowledged by a "Reject" message otherwise the node will assume as a parent the sender of the message, it will notify such decision by acknowledging with an "Accept" message and, finally, it will try to announce itself as the parent toward all the remaining neighbors.

The response to a "BeParent"message can be either "Accept" or "Reject" (see Fig. 3). In the first case the responding is added to the `children` list, while in the second case it is added to the `other` list. After each response, the node checks if all the neighbors have acknowledged its "BeParent" message and, if true, the node terminates its execution $\mathcal{JPDC}$ provides useful set operations such as `is_a_member` and `union` in the `NodeList` data structure.

3 $\mathcal{JPDC}$ Implementation

One of the peculiar characteristics in $\mathcal{JPDC}$ is that it provides a truly distributed programming environment[2] rather than using a simulation approach. $\mathcal{JPDC}$ has been implemented using the Java language, a natural choice suggested by its portability, its support for networking and by its wide diffusion and use in this environment [2].

The communication model is based on Message Passing implemented on the top of the JavaSpace framework from Sun. JavaSpace implements a shared, network-accessible repository for objects. Processes can use the repository as a persistent object storage and exchange mechanism; instead of communicating directly, they coordinate their activity by exchanging objects (i.e. messages) through the shared space. Processes can also subscribe to the Space so that they are notified when an instance of an object of the requested type is available into the Space.

3.1 Architecture

An application built using $\mathcal{JPDC}$ consists of several nodes (Java classes) that can be run everywhere in the network[3]. Three different kind of processes are present in $\mathcal{JPDC}$:

- `nodeProc`. By this (generic) name we mean the class that represents the computing node of the algorithm. For example, for the flooding algorithm described in Section 2 the `nodeProc` is `FloodingSTNode`.

- `shellProc`. This process (one per each `nodeProc`) is offered by $\mathcal{JPDC}$ to provide standard commands to control each `nodeProc`. All the `shellProc`s interact with a centralized `supershellProc` (one per algorithm) that allows to the user to set up network topology and send commands to all existing `nodeProc` processes. Several addressing options are available for communicating with only a subset of the existing `nodeProc`.

- `profileProc`. This process is in charge of gathering information describing the state of the algorithm execution. The obtained data can be useful both to profile the algorithms' performance both to log their activity and their status.

It should be noticed that the programmer only needs to write the relevant part of the `nodeProc` class (as described in Section 2).

3.2 Bootstrap of the application

At the start-up the user has to launch a Java Virtual Machine for each node (consisting of a `nodeProc` and a `shellProc`) on the desired hosts. Each node executes its *initialization* as specified into the constructor. Then, the nodes wait for messages from the `supershellProc` via JavaSpace. Then, the user needs to launch a `supershellProc` whose first step is reading the network topology and then broadcasting it to all the `nodeProc`s. Each `nodeProc` receives a unique ID and the adjacency list. It must be noticed that the step of the configuration is completely automatic and transparent, since the user is only required to write the configuration file. We must also add that it is possible to modify the network topology at run-time (adding/removing edges or deleting nodes) by using commands provided by the `supershellProc`.

After having built the network, the algorithm is ready to run. The user can choose a node (or possibly several nodes) where the computation is started by sending it the "execute" command through the `supershellProc`

[2]We believe that this choice is much more interesting because it allows users to better experiment the problems concerning with the real development of distributed algorithms.

[3]It means that the user can choose also to run the whole program locally, as well as (later on) distribute on several machines.

Pseudo-Code	Implementation

Response to receipt of BEPARENT from node Pj:
if parent is nil then
 	
parent:=j
send ACCEPT to Pj
 	
send M to all neighbors except Pj
else
 	
send REJECT to Pj
	case FloodingMsgType.BEPARENT :
	if (parent == null) {
	nmsg=new Message(getPid(), FloodingMsgType.ACCEPT);
	parent =j;
	send(nmsg, j);
	nmsg=new Message(getPid(),MsgType.BEPARENT);
	sendAllExceptOne(nmsg,j);
	} else {
	nmsg=new Message(getPid(),FloodingMsgType.REJECT);
	send(nmsg, j);
	}

Figure 3: Part of the *message reception handling* for Flooding algorithm. It describes the response to the receipt of BEPARENT message: the message is acknowledged either by replying with a ACCEPT (flooding also the neighbors) or a REJECT message.

process. This command starts the algorithm execution by invoking the method `execute()` (see Fig. 2).

3.3 Algorithm execution

After the *initialization* each `nodeProc` queries the Space for incoming messages (by subscribing to a particular instance of messages). A message (transmitted via the `send()` API) is represented by a `MessageEntry` object, each one containing a target field reporting the ID of the destination, the ID of the sender, a type and the message body. Moreover, each message contains also a unique sequence number in order to guarantee sequentiality of delivery. Every time a new message is put into the Space the target `nodeProc` receives a notification and, as a consequence, the new message is removed from the Space and put into an internal message queue maintained by the `nodeProc` according to its own sequence number. Then, messages in the queue are processed using the `processMsg()` method as defined (by the user) in the class `nodeProc`. The `processMsg()` is essentially a big `switch` where each `case` deals with a message type (see Fig. 3). The execution stops when each `nodeProc` explicitly invokes its `termination()` method or when the `supershellProc` is asked by the user to kill the node.

4 Interacting with the application

Testing and debugging a distributed application is, certainly, a capability that must be offered in a programming environment. At the same time, the opportunity to run a distributed application on a real distributed network, while providing the user with valuable information in a real setting[4], makes the task of controlling and testing a real challenge.

$\mathcal{JPDC}$'s `supershellProc` offers a graphical user interface that acts both as a (centralized) monitoring and control tool. It provides a graph-based visualization of the computing network where nodes are colored according to their status.The user can right-click on each node and have a report on the status of the internal (Java) variables.

The commands to control the execution of the distributed application are offered by the `supershellProc`, can be applied to all the nodes or singularly and can be grouped in three main categories:

- **Network topology commands** that can be used to change the topology of the network by means of link insertions/deletions/suspension/restorations. Additionally, it is possible to set the delay on a given connection link or to enable the logging of all the messages exchanged on one or more communication links.

- **Process control commands** that can be used to change the state of one or more nodes of the computing network by issuing "execute" or "reset" requests. In the same way, it is possible to know the state of each node of the network.

- **Introspection commands** that can be used to query at run-time the state of each internal (Java) variable of the nodes.

Moreover, $\mathcal{JPDC}$ provides a standard extension mechanism that can be used to easily extend the standard set of commands. Notice that, anyway, the same commands can be issued by interacting with the `shellProc` of each node. In fact, what the `supershellProc` does is send the commands to all the `shellProc`'s.

$\mathcal{JPDC}$ also offers a service for profiling the execution of the distributed application to an external log file that stores a trace of each significant operation performed by nodes. Profiling is accomplished by `profileProc` process (see Section 3.1) that gets from the JavaSpace all the "ProfileEntry" objects written by the nodes. The user is required (to get a more precise profiling) to use $\mathcal{JPDC}$'s `Status`-derived subclass to enclose the variables whose changes must be monitored. Profiling is an important capability: though computationally expensive, it allows precise post-mortem analysis.

[4]The "real" problems in a network (such as faulty nodes/links or nodes with different computational power) are unpredictable and are a real benchmark to the distributed application.

We also interfaced a typical $\mathcal{JPDC}$ distributed computation with the 3WPS visualization system[5] in order to provide a graphical representation of the behavior of a distributed algorithm.

5 Conclusions

Our motivating consideration to $\mathcal{JPDC}$ was to fulfil a gap between the general-purpose distributed programming frameworks and simulation environments for distributed programs. While the former (such as MPI and CORBA) are focussed toward the programmer that needs a detailed and technical view of the framework, the latter are aimed to education and training, therefore hiding some details to the "programmer" (such as simulating the real interaction with a network) and making impossible a real deployment of the realized implementation.

5.1 Related works

There exist several solutions to assist the user in implementing a distributed algorithm and understanding its behavior. In the rest of the Section we describe several programming frameworks or tools and fit them into three main categories.

Message-passing libraries. Programming frameworks such as CORBA or MPI provide the programmer with a wide choice of tools and efficient communication primitives. As a matter of fact, they require highly trained programmers and a deep understanding of the implementation details of the platform.

Simulators. A first (general) solution is offered by simulators like SIMC++ and SimJava[4] that allow to build working models of complex systems as networks of computing nodes. The execution is simulated by a single process while parallel execution is supported by a discrete event simulation kernel. A discrete event simulation makes it possible to use exact timings while defining node behaviors and communication delays thus allowing a very deep control degree on the algorithm execution. However, the main drawback of this approach is the high learning curve and the intrinsic complexity of the resulting system.

Prototyping tools. An alternative solution is offered by ad-hoc tools for the implementation of distributed algorithms such as HeNCE [3], designed to provide a software environment for developing parallel programs that must be run on a network of computers. In this environment any application program can be described by a graph. Nodes of the graph represent subroutines and the arcs represent data dependencies, individuals nodes are executed under PVM. HeNCE also provides a Graphical User Interface for creating, compiling, executing, and analyzing HeNCE parallel programs.

However, the most famous solution for implementing distributed algorithms in teaching environments is DAJ[6]. It provides an universally accessible platform for implementing distributed algorithms on an intuitive programming model. Such model uses the message passing paradigm: it abstracts a network of nodes connected by communication channel. DAJ operates in a single-process execution by simulating communication activity by means of internal process data delivering. The process executing a distributed algorithm can also be embedded in a Web browser as a Java applet.

$\mathcal{JPDC}$ can be placed in-between the message-passing libraries and the prototyping tools, being an effective way of engineering a distributed algorithm and, at the same time, producing a real distributed application. As a matter of fact, a wealth of algorithmic results waits to be transferred from the theoretical field to effective implementations. Especially in a teaching environments, complex algorithms can be much better understood and validated by implementing and running them. However, currently available programming frameworks for distributed programming are often too complex to use since they provide a low level framework and require good programming skills. Simulators suffer the drawbacks of offering only a centralized simulation of a distributed environment, that is (generally) much more unpredictable than any simulation.

$\mathcal{JPDC}$ fits into the prototyping tools category but offers the opportunity to deploy an easily implemented algorithm in a real distributed setting without compromising the flexibility offered to the programmer.

References

[1] H.Attiya, J. Welch. "Distributed Computing. Fundamentals, Simulations and Advanced Topics". MCGraw-Hill.

[2] G. C. Fox and W. Furmanski. "Java for Parallel Computing and as a General Language for Scientific and Engineering Simulation and Modelling". Proc. of 8th SIAM Conference on Parallel Processing for Scientific Computing. Minneapolis, Minnesota, March 1997.

[3] "HeNCE: Heterogeneous Network Computing Environmen". http://www.netlib.org/hence/

[4] F. Howell and R. McNab. "SimJava: A discrete event simulation library for Java". Proc. of 1998 International Conference on Web-Based Modeling and Simulation. Society for Computer Simulation International (SCS), January 1998.

[5] D.Malandrino, G.Meo, G.Palmieri, V.Scarano. "3WPS : A 3D Web-based Process Visualization Framework", Proc. of 1st IEEE Intnl. Symp. on 3D Data Processing Visualization and Transmission (3DPVT'02) June 19 - 21, 2002 Padova, Italy

[6] W. Schreiner. "DAJ – A Toolkit for the Simulation of Distributed Algorithms in Java". Technical Report 97-36, RISC-Linz, Johannes Kepler University, Linz, November 1997.