Dejay: Unifying Concurrency and Distribution to Achive a Distributed Java

Marko Boger, Frank Wienberg, Winfried Lamersdorf

Hamburg University - Department of Computer Science
Distributed Systems Group
Vogt-Kölln-Strasse 30, 22527 Hamburg
{boger, wienberg, lamersd}@informatik.uni-hamburg.de

Abstract

In the development of distributed programs two different concepts have to be considered, each being quite complex even on its own: concurrency and distribution. For both similar problems like synchronization and communication need to be addressed, yet are treated with completely different mechanisms. This paper presents Dejay, a programming language based on Java that unifies concurrency and distribution into the single mechanism of virtual processors. This allows a considerable simplification for the development of distributed or concurrent programs and makes the transition from a local to a distributed environment seamless.

1 Introduction

The most popular implementation language used for distributed systems is Java that, amongst other things, provides platform independence, a local concurrency model based on threads and a mechanism for remote method invocation (RMI). Compared to other languages like C++ it simplifies the development for a distributed environment by integrating these important mechanisms for concurrency and distribution directly into the core language. But on the other hand, concurrency and distribution need to be dealt with by using very distinct and unrelated mechanisms: Threads are only a mechanism to express concurrency on a single machine but do not allow to express the concurrency between remote machines. To deal with objects on remote machines, though, Java offers RMI, that transparently hides distribution. But it is not related to Java's concurrency mechanism. As a consequence, Java seems to be well suited for applet and client/server style programming, where the distinction between distribution and concurrency is very clear. An applet or a client can remotely access objects on the server using RMI, and concurrency within the server (or the client) is dealt with using threads. Indeed, Java is heavily used for this kind of programming.

In distributed applications beyond client/server architectures though, objects may need to migrate from one place to another. For example, if a producer creates an object hierarchy and needs to provide it to a consumer on a remote system, the provided object hierarchy might need to be migrated. Then the difference between remote access and concurrent execution becomes fuzzy. Java neither allows for migration of objects nor is it prepared to deal

with this relationship between concurrency and distribution. It does not sufficiently support the development of distributed applications with possibly migrating and concurrently executing objects. The Java community is so far stuck with client/server style programming.

Concepts better suited for distribution are needed. As a key to achieve this, the paper proposes to accept the challenges of concurrency and distribution by unifying them into a single concept. We first introduce this concept, called Virtual Processors, in section 2. These Virtual Processors control a group of related objects, allow their migration over networks and manage their synchronization. Then, an extension to Java called Dejay that implements this concept is presented in section 3. To demonstrate the simplified development using these technologies, a small example is presented. Finally, some comments on future directions and the relation to other projects as well as a conclusion are given.

2 A Mechanism for Migration

This paper discusses an abstraction from concurrency and distribution to a single unified concept. Currently, these two concepts are treated independently, which - at a first glance - may seem natural to most developers. Concurrency is used to execute different tasks independently and in parallel (or at least simulating it). For example, each of the five dining philosophers in Dijkstra's famous example can be represented by a thread, all executing on the same machine. On the other hand, techniques for distribution are used to communicate over a network. If, for example, each of the philosophers were implemented on a different machine, socket communication, RMI or CORBA-like architectures could be used to support the coordination of the philosophers' meal. Starting from any of these implementation possibilities though, a transition to a different one would result in a complete rewrite. In a system in which the philosophers could migrate from one node to another, the difference between distribution and concurrency could vanish. For example, all five philosophers could start out on one machine, executing concurrently, and then be migrated each to a different machine at an arbitrary point in time, now executing distributedly. Then a dining philosopher program could be written once and be executed either on one machine, all philosophers executing concurrently, or distributed over several machines, one philosopher per machine, or as a mixture of both or even change at runtime by migrating the philosophers from one machine to another, always using the same implementation. Therefore it is suggested to unify the concepts of concurrency and distribution by finding a proper mechanism for migration.

2.1 Virtual Processors

To achieve a unified view on concurrency and distribution we introduce the notion of Virtual Processors. A Virtual Processor can be seen as an abstraction of a single-threaded physical processor that controls objects in its address space and sequentially executes methods on these. The advantage of such an abstraction is that it is independent of its physical location so that a Virtual Processor can be migrated from one physical processor to another. A Virtual Processor can maintain connections to other Virtual Processors in such a way that these connections remain valid even if either of the Virtual Processors is migrated. This means that the physical location of a Virtual Processor is transparent and does not need to be known. However the location of a Virtual Processor can be changed. To express

concurrency, two or more Virtual Processors would be executing on one physical processor. To express distribution, Virtual Processors would be running on different machines. But since the physical location of a Virtual Processor is transparent, these two concepts are essentially the same. Since their physical location can also be changed by migration, each of these concepts can be transformed into the opposing one. Thus, the notion of concurrency and of distribution are unified by this approach.

For objects within each Virtual Processor the programming is as simple as usual single-threaded, non-distributed object-oriented programming. Imagine for example that Java had no threads and distribution mechanisms. A program written in such a simplified Java contains objects that reference each other and that can call methods upon other objects. Since there is only one thread of control, only one method is actively executed at any one time. This is just the way a program for a Virtual Processor works. The question to be answered now is how to connect objects to objects in remote Virtual Processors and how to migrate the latter.

Every object is contained in exactly one Virtual Processor. A reference from an object to another within the same Virtual Processor is an ordinary local reference as in regular Java. Objects in different Virtual Processors, though, can also be referenced, but in a different way. These remote references should syntactically be identical to local references but should be distinct in their semantics. A call along such a remote reference should of course retain the semantics of the equivalent local call but should respond to the different needs of distribution and remoteness. We therefore propose to introduce a distinct type for a remote reference that is syntactically equivalent but adds treatment of fault susceptibility, migration and larger response times and latency. Thus objects can be referenced in two different ways. A reference from an object to another object contained within the same processor is a normal reference. A call using such a local reference is executed in the usual way. A reference from one object to another one residing in a different Virtual Processor is distinguished: it has to be of a different type. This means that the difference between a local and a remote reference is expressed in the type system. Such a remote type is generated automatically by a compiler from an existing class. In the system described in the next section, such a type is distinguished from the local type by an additional prefix which is in this case 'Dj' so that the remote type for a class A would be 'DjA'.

Now, while the syntax of calling a method is the same for local and remote references, their semantics is not. A call using a remote reference is redirected to the remote Virtual Processor by a proxy mechanism completely transparent to the programmer. There, the incoming call is queued until all other execution in this processor has terminated in order to keep execution strictly sequential within a Virtual Processor. It is then executed and the result is sent back to the calling party. A remote call can be either synchronous or asynchronous. If it is asynchronous the remote processor will execute this method in parallel to the calling processor so that the calling one can continue with other tasks. This is the means to express and create concurrency.

Virtual Processors are also the unit of migration. Objects with tight couplings can be assigned to one processor, making this concept a grouping mechanism for objects with the processor as the execution unit of each of the resulting components. When the Virtual Processor is migrated all objects controlled by it migrate together with it. Therefore, in our terminology a Virtual Processor is the unit of migration. The granularity of migration is determined by the number and complexity of contained objects and can therefore be designed and controlled by the programmer. He can choose to either instancitate exactly

one object per Virtual Processor, so that the migration granularity is as fine as possible, or he can put a whole application into a single Virtual Processor and move it as a whole, or anything inbetween. Most typically he would put closely related objects or objects that need to be colocated in the same Virtual Processor and keep loosely related object in different ones.

This concept is similar to suggestions by [6] proposed for the programming language Eiffel. Meyer shows that this concept integrates well with object-orientation, synchronization and inheritance but has so far not provided an implementation of this concept. Also he proposes to introduce a new keyword to distinguish between remote and local references while we propose different types. While Meyer has to introduce additional rules when and how this keyword is needed or forbidden, our solution smoothly integrates into the existing type mechanism so that the usual type conformance checking makes additional rules obsolete.

3 Dejay - A Language for Distribution

The following sections present a language that integrates the concept of Virtual Processors into the Java language to achieve a distributed Java. This language is called Dejay. It aims at simplifying the modeling, development and the programming of distributed systems and allows the execution in an adequate concurrent and distributed way. The assumption is that the environments found today or in the near future are sufficiently reliable and fast to establish distributed applications. Such environments may be the Internet, Extra- and Intranets as well as local area networks.

Dejay is not designed to be a completely new language but an enhancement of Java for concurrent and distributed programming so that the syntax of Dejay is very similar to that of Java. In fact, in case of single-threaded non-distributed applications Dejay is identical to Java. But for multi-threaded or distributed applications Dejay is mostly a subset of Java: the threading mechanism of Java as well as the remote method invocation mechanism RMI have completely been replaced and all keywords concerning threads are not needed in Dejay. This reduces the complexity of the language considerably.

Instead of these, Dejay introduces the concept of Virtual Processors. Every object is embedded in a Virtual Processor and is under its exclusive control. If only one thread of control is needed, the concept of Virtual Processors remains completely transparent. Only if concurrency or distribution are required, further Virtual Processors are created. No additional keywords are introduced in Dejay, however the type system as well as the semantics of references and method calls are changed, so that a compiler is needed that translates Dejay to Java code.

3.1 Creation of Virtual Processors

The most important new concept of Dejay is the Virtual Processor. It can be started on the same or on some other remote machine that is reachable over an IP-network and, since the implementation of Virtual Processors relies on Voyager [7], has a Voyager daemon running on a known port. In the following we will use an abbreviation for the IP-address and the port number of such a daemon, such as `134.100.11.185:8000`, and simply call this `X`, `Y` or `Z`.

In Dejay, the equivalent to spawning off a new thread is to create a new Virtual Processor

on the same machine. This is simply done by creating an instance of the class `Processor`. The preceding "`Dj`" is explained in the next section.

```
// create a processor on local machine X
DjProcessor p1 = new DjProcessor();
```

A typical use of this would be within a chat- or http-server that needs to handle new incoming requests in concurrently executing handler classes. In Java this would mean spawning off a new thread, but while in Java this can only be done on the same machine, in Dejay this can be done on either the same or on any remote machine. To create a Virtual Processor remotely its constructor is simply passed the name of the intended machine.

```
// create a processor on a remote machine Y
DjProcessor p2 = new DjProcessor(Y);
```

3.2 Local and Remote References

The second important concept is that of local and remote references. We believe that a distinction between an object that is local and an object that is remote should be made at all times for two reasons. Firstly, the time delay of a local and of a remote call can differ in several orders of magnitudes so that it is important to differentiate between local and remote calls already at design time. Secondly, remote calls require a different treatment to local calls, such as extended exception handling or asynchronous message passing. To make this distinction we extend the type system of Dejay in comparison to Java: a reference to a local object has a different type than a reference to a remote object. Of course these two types have a very similar signature and fulfil the same functionality, but they should be incompatible and express the difference at any time. In this way the semantics of a remote call can be changed while maintaining the same syntax.

To differentiate these two types of references, the type names of remote references are constructed from the usual local type name (to express the similarity) but preceded by the letters "Dj" (to express the difference). For every class compiled by the Dejay compiler `djc`, class definitions for both types are generated. An alternative to this is to introduce an additional keyword as proposed by [6] that marks remote references and to leave type names unchanged. Then, however, additional assignment rules need to be introduced since local and remote references need to be incompatible in the first place. We have rejected this approach to integrate our concepts as smoothly as possible into the existing Java language and its type system.

Objects on the local Virtual Processor are created and used in the usual Java way. Assume for example an object `a` of class `A` instantiated in Virtual Porcessor `p1`, located on machine `X`. To create an instance `b` of class `B` on the same Virtual Processor, the code of class `A` may contain the following :

```
// create an object on same processor
B b = new B();
b.some_method();
```

The creation of instances on remote Virtual Processors is done with the Dj-type of a class. It does not matter whether a Virtual Processor is on the same or on a different physical machine, for this difference is abstracted away by Dejay (and the location can change at

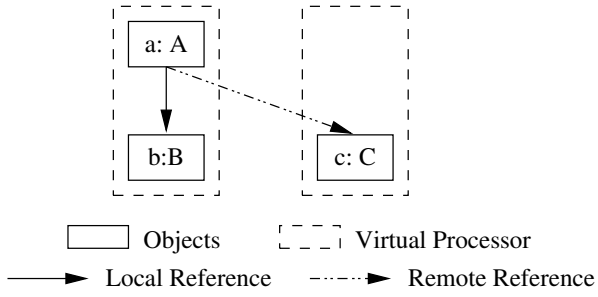

Figure 1. Local and Remote References

runtime anyway as we will see later). To create an instance of, for example, `C` on a remote processor on a (potentially) different physical machine one would write the following:

```
// create an object on processor p2
DjC c = new DjC(p2);
```

This results in a situation shown in figure 1.

3.3 Migration

A Virtual Processor can be moved from one machine to another simply by calling a `moveTo()` method. As an argument this method accepts an IP address, a host name or a reference to another Virtual Processor. It then migrates the complete Virtual Processor (and all contained objects) to the specified machine or the machine running the specified Virtual Processor, respectively, and leaves a forwarder behind, so that calls to this Virtual Processor will be redirected to the new location. If no argument is given, it moves to the machine of the calling object. Calling the `moveTo()` method on an object contained in a processor will result in the movement of the entire processor as well.

```
// move processor p1 to machine Z, including its object a and b
p1.moveTo(Z);

//equivalent: move a to Z, including its processor p1 and object b
a.moveTo(X);
```

Logically the situation has not changed, all references remain valid. But physically a component has migrated at runtime as shown in figure 2. Migration becomes simple and secure. Objects are always moved as a group, keeping related objects together avoiding the extra step of analyzing an object's closure. Communication between objects belonging to different processors is the same whether the processors reside on the same or on different machines. But moving the processors to a local machine can reduce communication time tremendously.

3.4 Concurrency

The concept of processors is also used to express concurrency. Concurrency can be used to perform actions in parallel, which requires several physical processors. Or it can be used to control different threads of control on one physical processor, only simulating parallelism, for example to have one thread calculating while another is waiting for input. If two Virtual

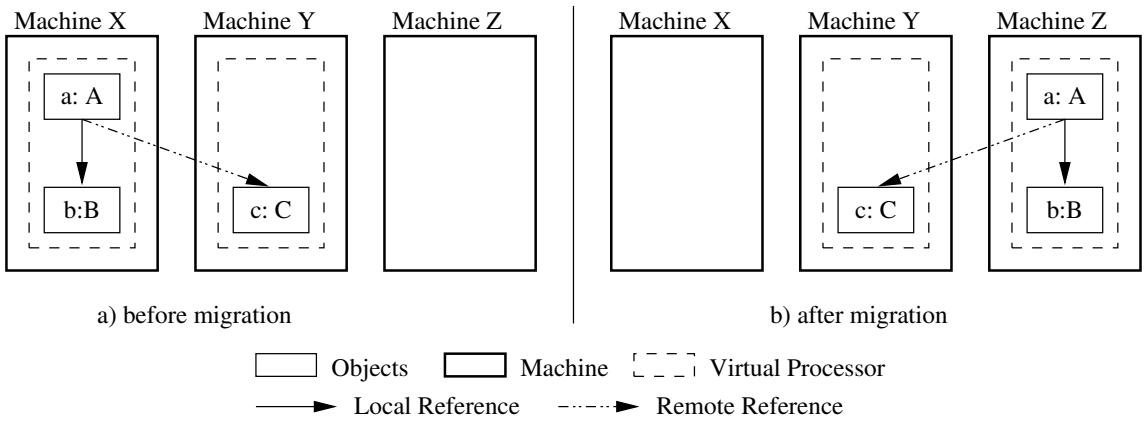

Figure 2. Physical situation before and after migration

Processors run on the same machine, parallel execution is only simulated. If one of them is moved to a different machine, real parallelism is exploited.

In Dejay method calls can be synchronous or asynchronous, while in Java method calls are only synchronous. Calling a method on a remote object, i.e. using RMI, may result in long waiting times since the calling object is blocked until the result is returned. To simulate asynchronous behavior in Java, a new thread needs to be spawned off to handle the call and await the result. This makes asynchronous calls tedious and error-prone. But a simple mechanism for asynchronous calls is vital for distributed programming since parallelism on remote machines can be achieved by asynchronous calls. Therefore, Dejay incorporates and facilitates the use of asynchronous calls.

By default, Dejay executes calls to remote objects in a synchronous fashion. The thread of control is passed to the remote processor hosting the called object and is handed back when the call returns. However, it is also possible to issue an asynchronous call to a remote object that is executed in parallel and therefore spawns off a new thread of control. Such a new thread rejoins the original one when the result is returned. As a third possibility, one-way method calls, can be used to omit the rejoining phase if the method's result is not needed.

To distinguish between these cases, a method call can contain an additional parameter. This parameter specifies whether the call is a synchronous, an asynchronous or a one-way method call. The argument is a constant of type `Dejay.base.Messenger` object. Three different types of messengers exist that determine the semantics of the call, respectively:

```
// synchronous, blocks until the result is returned
// this is the default and Dejay.SYNC can be omitted
result1 = c.some_method(Dejay.SYNC);

// asynchronous call, continues immediately
result2 = c.some_method(Dejay.ASYNC);

// one-way call, continues immediately, no rejoining of threads
c.some_method(Dejay.ONEWAY);
```

To rejoin an asynchronous call and to use the returned result, *wait-by-necessity* is used. That

means that an application would automatically block when the variable the result is assigned to is first used. To explicitly wait for the result to return, a method `Dejay.wait(result2)` can be called. It can also be tested if a result has returned by calling `Dejay.poll(result2)`, which returns a boolean value so that the programmer can decide whether the program should wait or continue.

3.5 An Example

To demonstrate Dejay a small example is given. Consider the following class HelloWorld.

```
public class HelloWorld{
   public void sayHello(String s) {
      System.out.println("Hello "+s);
   }
}
```

Syntactically this class differs in nothing from a conventional Java-class. It is simply compiled with the Dejay compiler `djc` to produce a distribution-, concurrency- and migration-aware class.

This class can be instantiated like any other Java class. But to demonstrate Dejays abilities, an instance shall be created remotely and called so that it will print `Hello Ping` on the screen of a remote machine `X`. Then the object is migrated to a different machine `Y`. Called again, it will put out `Hello Pong` on that machine.

```
public class Startup {
   static void main(String[] args) {
      //First, say hello on machine X
      DjProcessor p1 = new DjProcessor(X);
      DjHelloWorld hello = new HelloWorld(p1);
      hello.sayHello("Ping");

      //Then, say hello on machine Y
      hello.moveTo(Y);
      hello.sayHello("Pong");
   }
}
```

We believe that this is the shortest and, more importantly, simplest way to express this kind of functionality in an object-oriented, java-like fashion. The advantages are even bigger if the application is more complex and a virtual processor contains more than one object. Then whole components can easily be moved around a network with all local and remote references automatically retained.

4 Future Work

The implementation of the presented work has reached a stable state. The next step is a closer evaluation of the runtime behavior and some fine tuning. A synchronization mechanism to express wait conditions is currently being developed. The distribution of objects

in space, as presented here, and the distribution of objects in time through persistence hold similar problems. The potential of Virtual Processors as a clustering mechanism for persistence is currently being evaluated. The presented paradigm also appears to be very suitable for independent self-triggered migration as mobile agents do. Since a Virtual Processor is currently implemented as a Voyager object [7] and since Voyager offers a mobile agent mechanism, a Virtual Processor can also be seen as a mobile agent.

Another important aspect of distributed program development is modeling. Standardized modeling methods like UML give very little or no support to model the distributed or concurrent aspects of a software system. The main reason of this might be that the current languages only support relatively complex techniques that are difficult to represent in a model. Since the concepts used in Dejay allow a more abstract view on a distributed and concurrent system, they are more suitable for modeling such systems as well. A modeling technique as well as supporting tools that extend UML integrating these concepts are currently being developed.

5 Related Work

It has repeatedly been discussed that Java, as is, is not very well suited for distribution [5]. Especially the current mechanism for remote method invocation has widely been criticized [8]. Therefore there is a great interest in Java-based or Java-extending solutions that aim at improving the distribution abilities of Java. A number of projects try to achieve this by providing better libraries or frameworks for distributed communication (Voyager, iBus, JavaGroups, Java ACE, Habanero, JSDT) or by modifying the existing RMI mechanism (NinjaRMI, JavaParty, FarGo). Some projects develop languages similar to Java (Infospheres, Pizza, JavaParty) or provide a modified virtual machine (DJ, Pjama). Others incorporate new communication paradigms like Agents (Voyager, Odyssey, Mole, Straum) or tuple spaces (JavaSpaces, TSpaces).

Many other projects exist that examine distribution or concurrency concerns for object oriented languages. Emerald [2] had a strong influence, for example on DOWL [1] or Beta [9]. Here, migration of groups of objects is expressed by explicit attachment. Its relation to Dejay is discussed in [3]. For a recent overview on concurrent OO-languages see [4].

6 Conclusion

This paper shows that the development of distributed systems does not have to be as hard as it is today, where matters of concurrency and of distribution have to be dealt with using completely distinct techniques. An abstraction unifying concurrency and distribution into a single concept and allowing for transparent object migration was described. Dejay, a new programming language based on Java, that implements this concept, was presented. Dejay expresses distribution, grouping and migration of objects and concurrency using the concept of Virtual Processors. It assembles a set of closely related objects and defines a granularity of migration. Within this, execution is sequential while other Virtual Processors execute concurrently to it. The unification of concurrency and distribution into a single concept makes the development of distributed systems simpler as compared to existing solutions like Java RMI, CORBA, or Voyager.

References

[1] B. Achauer. The dowl distributed object-oriented language. *Communications of the ACM*, 12(9), September 1993.

[2] A. Black, E. J. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in emerald. *IEEE Transactions Software Engineering*, 13(1), 1987.

[3] Marko Boger. Migrating objects in electronic commerce applications. In *Proceedings of Trends in Distributed Systems for Electronic Commerce*, 1998.

[4] J. Briot, R. Guerraoui, and K-P. Löhr. Concurrency, distribution and parallelism in object-oriented programming, December 1997. Technical Report B-97-14, FU Berlin, FB Mathematik und Informatik.

[5] G. Brose, K.-P. Löhr, and A. Spiegel. Java does not distribute. In *Proceedings of Technology of Object-Oriented Languages and Systems TOOLS Europe '97, Paris*, 1997.

[6] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, second edition, 1997.

[7] ObjectSpace. Voyager, 1999. www.objectspace.com.

[8] Michael Philippsen and Matthias Zenger. Javaparty - transparent remote objects in java. *Concurrency: Practice and Experience*, 9(11), November 1996.

[9] O.L. Madsen S. Brandt. Object-oriented distributed programming in beta. In *Proceedings of Object-Based Distributed Programming, ECOOP'93 Workshop, Kaiserslautern, Germany, Lecture Notes in Computer Science, Vol. 791, Springer Verlag*, 1993.