Virtual Processors: Migrating Object-Clusters unify Concurrency and Distribution

Marko Boger, Frank Wienberg, Winfried Lamersdorf

2nd February 2000

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

Abstract

Concurrent and distributed software systems are currently very distinct in their usage and programming. In the case of Java the first requires threads, the latter RMI, CORBA, Voyager etc. However, they are tightly related and often cited in one breath. This paper presents a concept that unifies these two aspects into one. To achieve this the concept of virtual processor, a mechanism to cluster and transparently migrate groups of object, is introduced. It can be used to express concurrency as well as distribution and migration can turn one into the other. As a proof of concept a dialect of Java that implements this concept and some examples are presented.

1 Introduction

Why is software development for a distributed environment so hard? Developing a single-threaded local application can already be very difficult. But the world of programming has changed and today we find ourselves programming for concurrent and distributed environments. Concurrency requires the handling of multiple threads of control, synchronization, deadlock avoidance and the like. Debugging in a multithreaded environment becomes close to impossible. For distribution, different mechanisms for network communication, object migration, security etc. need to be introduced. The development of distributed applications is often considered to be one of the most difficult tasks in software development. In particular, distribution often leads to concurrency due to the decoupling of communicating entities.

One of the most popular implementation languages 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 concurrency between remote machines. To deal with objects on remote machines, though, Java offers RMI. 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 wants 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

One of the key mechanisms for the improvement of computer science is abstraction. Three examples of abstraction that have fundamentally improved software development are a) the abstraction from data and functions to objects, b) the abstraction from hardware tasks and processes to virtual software threads and c) the abstraction from different incompatible hardware platforms to a virtual machine as in Java.

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 rewrite of larger parts of the application. 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.

The mechanism proposed in this paper is based on Java. Principally, the abstractions proposed are not restricted to this language, but Java provides the prerequisites necessary, especially for migration in an easy way. Java itself does not support object migration, but extensions exist that do. One of them is Voyager.

2.1 Migration in Voyager

Voyager [Obj98] is an ambitious project that aims at providing a complete infrastructure for distributed programming, integrating or even replacing other techniques like RMI, CORBA, DCOM and Agents in general. It is based on and completely written in Java and offers compiler and runtime mechanisms that prepare any Java object for distributed computing. Besides propositions for autonomous migration, persistence, security, CORBA-integration and multicasting, Voyager introduces the following approach for referencing and migrating remote objects.

Voyager supports object migration by supplying a `moveTo()` method for a proxy of a remote object. This method - when called with an IP address or host name as an argument - will move the referenced object to the specified machine. Assume we have an object `a` of class `A` on machine `X` as shown in figure 1. It holds references to an object `b` of type `B` and to a proxy `c` of an object of type `C` on machine `Y`. Further, an object `d` of type `D` holds references to objects `b` and `c`.

Using Voyager, the object `a` can be moved to machine `Z` by sending it the method call `Mobility.of(a).moveTo(Z)`. The question is what happens to references and referenced objects if `a` is moved? Figure 2 shows the situation after this migration. Object `a` itself leaves a proxy behind and migrates to `Z`. If object `d` had a reference to it, it would now reference it over a proxy. The proxy object `c` is simply copied to machine `Z` without affecting the rest of the system (see object `c` in figure 2). Since object `a` has a reference to a local object `b` of type `B`, `b` needs to be copied too, to assure correctness of the references of `a`. In fact, Voyager copies all objects reachable from `a` via local references, thus copying what is called the transitive closure of `a`. But if `a` accesses the copy of `b` and changes the state of it, this change does not affect the original copy of `b`, referenced by object `d`. Thus the two copies of `b` are not in a consistent state. Also, if `b` contains references to further objects which will be copied with it, the amount of data being moved can be enormous and is difficult to control. In order to avoid these problems, all references from an object that is to be moved need to be checked by the programmer and, if necessary, either be cut, or replaced by a reference over proxies. In praxis, this often means that only objects with very small transitive closures or even with no references at all can be moved.

Concluding, it can be noted that Voyager does offer object migration to a certain extent but it has to be used with great care. The migration of an object results in copying its transitive closure which can a) lead to inconsistent states of different copies of an object and b) result in the movement of vast amounts of data if references from the object that is to be moved to objects, that should not be, are undetected. It is the programmers responsibility to avoid these problems and therefore, the development of distributed systems based on Voyager still remains an error-prone task. The mechanism proposed in this paper is built on top of Voyager mechanisms, but at a higher level of abstraction, and avoids these problems, as explained in the next section. Most important though for the discussion in this paper is that Voyager does address aspects of distribution such as migration, but does not touch on matters of concurrency. Thus it is not the abstraction sought for. However, it is very helpful in the implementation of such an abstraction.

2.2 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, this approach treats concurrency and distribution as just two sides of a coin.

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 point in 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 automaticredirected to the remote Virtual Processor by a proxy mechanism. 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. The granularity of migration is determined by the number and complexity of contained objects and can therefore be designed and controlled by the programmer.One can choose to either instantiate exactly one object per Virtual Processor, so that the migration granularity is as fine as possible, or one can put an application into a single Virtual Processor and move it as a whole, or one can choose anything in between. Most typically, one would put closely related objects or objects that need to be co-located in the same Virtual Processor and keep loosely related object in different ones.

Consider again the scenario given in the last section where Voyager proved to have difficulties, especially with migrating groups of objects. Object `a`, `b` and `d` were located on one machine, where `a` needed to be moved and `b` was referenced by both others. Also, `a` and `d` held a reference to some object `c` on a different machine. By introducing Virtual Processors, the notion of machine can be abstracted away, however the notion of remoteness remains. The remote object `c` will clearly be put into a distinct Virtual Processor. Whether this is actually placed on machine `Y` or moved there later or even removed from there does not matter. It is referenced by `a` and `d` remotely, but its physical location is transparent to them. Since `a` is to be moved and `d` is to remain, they should be placed in different Virtual Processors, even though they might start out on the same machine. It needs to be decided though where `b` is put. By analyzing its relations, it should be decided whether it is related more closely to `a` or `d` and put in the same Virtual Processor. In either way, both could reference and access `b`, one locally and the other remotely. If it is found to be closer to `a`, it would migrate with it when `a` was moved and `d` would not even need to notice. See figure 1; note that it depicts the situation before just as well as after a migration, since this remains transparent.

This concept is similar to suggestions by [Mey97], 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 . 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 modeling, development, and programming of distributed systems and allows execution in an adequately 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 which translates Dejay to Java.

3.1 Application Scenarios

Before the syntax and usage of Dejay is described, some practical application scenarios should be helpful. Dejay is part of a research project called COSMOS [COS98], funded by the European Commission. The aim of the COSMOS project is to provide a distributed contracting environment for electronic commerce over the Internet and, due to the distributed nature of such an application, has very high demands on the distribution support of the implementation language. Dejay is being developed as an implementation language within the scope of COSMOS to meet these demands. In the following, two application examples from within

COSMOS are sketched, where common technologies are not sufficient and a higher level support is required.

Generally speaking, the COSMOS project provides an infrastructure for establishing electronic contracts over the Internet. It is divided into different phases: In a first stage, a contract proposal can be established from a catalog of offers, service providers and contract templates. Once such an offer is set up, the proposed contract can be negotiated on by the involved parties. It is important to note that more than two parties can be involved, for example for the leasing of a car, a car seller, a buyer that is interested in a car, a notary who will check legal validity of the contract, and a bank taking financial risk and giving a loan could be involved. After reaching agreement with all parties, the contract is signed and stored for legal purposes at the notary. In addition to usual written contracts, electronic contracts can also be used to support the execution of a contract. In COSMOS, a contract contains a workflow definition that can be executed as a distributed workflow application.

The first example is taken from the negotiation phase during which a contract proposal can be changed in terms of price, conditions or legal constraints by either of the involved parties until all agree. This requires the editing of the electronic document (represented as a Java-object structure) as negotiation proceeds. The simplest way to avoid inconsistency during editing, is to only allow one single party to edit the contract that is placed on a central server and can be locked, copied to the requesting party, edited and copied back

This can be extended in two ways to better reflect the needs and conditions of reality: on the one hand, more than one editor at a time should be allowed, so that, for example during a multimedia conference, all negotiating parties can write propositions into the contract in a way that all others can see the changes. On the other hand, the system can be extended by distributing control: The master copy of the contract can be migrated from one host to another, for example from the broker that sets up the contract proposition to the editing party and further to the notary, where it is finally stored, but needs to be accessible from all parties. This results in two orthogonal and discrete dimensions (single party/multiple party access and centralized/distributed management) and four possible combinations, as shown in table 1.

The first combination (single party access/centralized (1)) represents the simplest solution and can be implemented using standard client/server technology with locking. If the contract is allowed to migrate (single party access/distributed (2)) a technology like Voyager which allows for simple object migration is required, but locking has to be taken care of manually or by an additional transaction service (to be published for Voyager in the next version). If multiple parties are allowed access at the same time (multiple party access/centralized (3)), consistency control becomes considerably more difficult since it requires fine grained locking or conflict resolution mechanisms. Nevertheless this can be solved with usual client/server technology, but requires more effort for the locking.

But in the case of multiple party access with distributed data (4), current technologies fall short. However, this is the most realistic setting for this application. The contract object model should be movable from the contract provider to one of the involved parties so that editing can be done locally. At the same time other parties should have access to it to read or even to change it, as for example in a multimedia session (see figure 4). The goal is a location transparent access where the location of the contract can be changed, for performance as well as for responsibility, liability or ownership reasons, but all references remain valid, even if the contract is moving.

The second example is taken from the execution phase of COSMOS, in which a contract is being executed as a workflow. Such a workflow can automatically be extracted from the contract object structure and is expressed in a Petri Net-like graph as described in [COS98]. Now, usually workflows are executed on a central server by a workflow engine that can access and call subsystems engaged in the workflow, for example using RMI. This works well in case of single server systems or small networks. But in the setting of COSMOS the engaged subsystems are part of distinct, potentially very distant networks of participating parties that only interact spontaneously so that a client/server architecture falls short. What is needed is a distributed architecture where objects can be created remotely, objects can be migrated and references are location transparent. The approach followed in COSMOS is therefore to distribute the application instead of distributing the communication between workflow engine and the distributed parties. Each transition of the Petri Net graph represents one of the involved parties and the interaction between this representative and the actual software system of the parties host can be very intensive, while the interaction between Petri Net nodes is scarce and unidirectional and consists of the transmission of (object-) token. Thus the workflow engine itself is distributed, and the most costly parts of the communication can be done locally at the distributed hosts.

For the example given above such a workflow is shown in figure 5: the involved parties are the seller, the buyer, the bank and the notary. For each of these a representative (a business object) is set up at the parties host. Then objects that describe the activities each party is responsible for are created, their relations established and finally moved to the representative. The loan-activity then interacts with the bank and initiates the transmission of a loan to the buyer, informs the observe-activity of the notary and triggers the pay-activity of the buyer. This object then interacts with the buyer and arranges the payment to the seller, informs the notary and triggers the deliver-activity of the seller. This object then induces the delivery process on the sellers software system and so forth.

This shows two applications that the Dejay language is being designed for that are very distinct in their nature and show the high requirements that need to be fulfilled: In the first example, a unique object cluster, the contract description needs to be accessed from several distributed clients in a location transparent way. In the second example, the application itself, the distributed workflow engine, is distributed, in order to reduce communication costs. Both require object migration. Current Java technology falls short in the described settings. A deeper discussion and further application descriptions are given in [Bog98].

3.2 Compilation

Dejay is a programing language and therefore has a compiler that currently produces Java source code as output. Dejay source files have the ending `.dj` so that a class `A` is stored in a file called `A.dj` and can be compiled using the compiler `dejayc`.It temporarily produces two output files, a file `A.java` and a file `DjA.java`, that are in turn compiled by an ordinary Java compiler to `A.class` and `DjA.class`. At a first glance and for simple (single-threaded non-distributed) programs Dejay appears to be identical to Java so that, for example a "Hello World" program looks the same.

```
public class A {
   public A() {
   }
   public void m1(String s) {
      System.out.println(s);
   }
   static void main(String[] args) {
      A a= new A();
      a.m1("Hello World");
   }
}
```

At a closer look though, there is more behind the scenes. In Dejay every object is embedded in a Virtual Processor, however, the first Virtual Processor in a distributed program is created automatically so that a class `A` can be started as usual using its `main()` method (which is manipulated by `dejayc`) and does not need to be aware of Virtual Processors at all.

3.3 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 [Obj98], 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);
```

Also a virtal processor can be created "close" to some other object. By passing a reference as an argument to the constructor a new Virtual Processor can be created on the same machine as some known processor or some other known remote object. Note though, that the referenced object will not be within the new processor but simply on the same machine.

```
// create a processor on the same machine
// as some known object obj
DjProcessor p3 = new DjProcessor(obj);
```

The mapping of virtual processors to physical machines is very flexible. It can either be done statically by hard-coding the corresponding strings. Or the configuration can be read in from a file at startup and assigned to variables like `X` and `Y`. Or it could even be changed at runtime for example by a tool that could monitor and reconfigure the physical placement using migration, as shown further down.

3.4 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 `dejayc`, class definitions for both types are generated. An alternative to this is to introduce an additional keyword as proposed by [Mey97] 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 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);
```

To obtain a remote reference to a local reference, one can call the constructor of the Dj-class with a reference to the local object. These remote references can be passed as arguments across processor boundaries and used as arguments for methods or constructors. To demonstrate this, a new object `d` is created in the existing processor `p1` (that happens to be on the same physical machine) and passed as a remote reference to `b`, once using the constructor and once using a method.

```
// create a remote reference
DjB rem_b = new DjB(b);
// pass reference in constructor ...
DjD d = new DjD(rem_b, c, p1);
// or pass reference as parameter in method call
d.setB(rem_b);
```

The code presented so far will result in a similar scenario as described in the previous section. It is shown in figure 6, including physical locations of each Virtual Processor.

3.5 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 or remote object. 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(Z);
```

Logically the situation has not changed, all references remain valid. But physically a component has migrated at runtime as shown in figure 7. 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.

The unit of migration is the Virtual Processor and not the object. Objects can not be migrated out of a virtual processor in that sense, that they can not move while keeping their identity. However, they can be copied from one to another Virtual Processor and if the original is dropped it can be thought of as migration. But the copy has its own identity and references to the original are not redirected.

The migration mechanism is implemented based on voyager. The Virtual Processor is a voyager object and is migrated using the voyager mechanism discussed above. However, since every object belongs to exactly one Virtual Processor, the problems that appear using Voyager directly are avoided.

3.6 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 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 `result2.wait()` can be called. It can also be tested if a result has returned by calling `result2.poll()`, which returns a boolean value so that the programmer can decide whether the program should wait or continue.

3.7 Synchronization

In Dejay each object belongs to and is managed by a component controlled by a single Virtual Processor. Objects can be called from outside the processor by using their virtual objects.

But the processor intercepts incoming calls at the components boundary, queues them, and executes calls to its internal objects in a sequential manner. This greatly simplifies synchronization by reducing the need for it. No synchronization is needed between objects contained in the same processor. Objects are encapsulated within the Virtual Processor in which concurrent threads are prohibited so that they are always used exclusively. Synchronization between objects in different processors is much simpler than in Java. Since a call to a remote object is always redirected to the encapsulating Virtual Processor and queued until a currently executing method is terminated, it cannot interfere with other operations and change the state of an object unexpectedly. In fact, locking of objects is only necessary in a transactional sense if an object is required in an unchanged state over two or more remote calls. We are currently working with an extension of the Java keyword synchronized that accepts not only one but several objects as arguments and blocks all of these until the synchronized-block is terminated.

This allows the efficient support by the compiler translating these high level constructs to low level Java synchronization. Similar to the replacement of pointers in C++ by references in Java, or the use of garbage collection instead of explicit memory allocation, the replacement of low level synchronization mechanisms like semaphores by the more abstract view of complete Virtual Processors simplifies the language and allows automatic generation of efficient code. Different mechanisms for expressing synchronization constraints are currently being investigated, including that of Eiffel [Mey97] using pre- and post-conditions as wait conditions. Also, separate, explicit synchronization specifications or synchronizers as discussed in [Frø96] are explored. An important aim is to avoid the inheritance anomaly discussed in [MY93]. An inheritance anomaly can occur when synchronization is defined in a superclass and redefined in a subclass. This can completely mislead synchronization.

3.8 Exception Handling

Exception handling is not fully implemented yet but basic mechanisms exist. Exceptions that are thrown by the remotely referenced object are transmitted to the remote caller and must be treated in the usual way. Thus the semantics of a class for exceptional cases is not changed. But additional errors can occur due to the fact that the call is remote. This can be network failures, time-outs, etc. and a programmer needs a possibility to react on these. However, here we are facing the decision between an optimistic and a pessimistic approach. RMI for example represents a pessimistic approach, where for every remote call remote exceptions must be caught. This clutters up the code and makes it more difficult to read and develop. A more optimistic approach is to catch the exception on the level of the proxy, and to re-throw it as an error. In Java errors can be but do not have to be caught. This way the developer can decide if he wants to take care of network failures or if he wants to keep the code simple as for example in prototypes.

3.9 Implementation

The Dejay language has successfully been implemented and exists in a consistent and demonstratable version. However, the language is still under development and will further be extended. Its implementation has two parts. Firstly, the concept of Virtual Processor was implemented by an according Java solution. Secondly, a compiler that translates Dejay to Java was developed.

For the implementation of the concept of Virtual Processors a number of different alternatives have been evaluated. Of course, this could have been done using the Java JDK alone but this would have meant reinventing the wheel since there exist solutions this project can rely on. For the evaluation of distributed extensions to Java a list of criteria has been developed that needed to be fulfilled to successfully implement Virtual Processors. The criteria were

- a) a basic support of migration of objects. It should be possible to move an object without loss of its identity and relation to other objects.
- b) the ability to create proxies of any given object at runtime. Many mechanisms only can create proxies at compile time, which is not sufficient.
- c) a possibility to access and interrupt the flow of control when a remote call comes in. To adequately administer a running system the Virtual Processor needs to be informed of incoming calls to delay the call and additional actions. It may not just be bypassed.
- d) an optional type compatibility of remote references to normal references for compatibility to usual Java. Since not all Java classes can be rewritten in Dejay, the remote reference type needs to be compatible to what normal Java classes are used to.
- e) access to local references of a remotely created object on the remote machine. This is needed to support the serialization of a distinct but intertwined object graph. It is necessary to create a single root for all objects in question for migration.

A number of different techniques has been considered, most closely RMI, JavaParty, CORBA and Voyager. None of the evaluated techniques supported these features directly. RMI and CORBA already fall short of the first. The option that finally proved most suitable was Voyager 2, however, it did not fulfill all criteria, namely c) and e) so that it was not possible to use it directly. The Virtual Processor is implemented as a Voyager object, but a mechanism to implement proxies to remote objects had to be constructed from scratch using Java Reflection.

A Virtual Processor is implemented and described by the class `Processor`, sketched in figure 8. It maintains a list of all objects that are referenced remotely and contained within it. New objects that are created from remote are automatically registered to it and de-registered when their proxies are garbage collected. It also maintains a queue of incoming calls. It dispatches calls one at a time, retaining other calls until the dispatched call is done and the result is returned. Thereby it insures a sequential processing of the functionality offered by the sum of the contained objects.

The use of Proxies is hidden from the programmer. To him remote objects are simply referenced over a different type than a local object. However this is implemented using proxies that represent the remote object on a remote machine and offer a superset of the original interface.

Assume an object `obj` that holds a remote reference to an object of type `A`, thus a reference of type `DjA`, shown in figure 9. `Obj` can invoke a method of `A` which will be handled by the proxy (step 1).

The proxy does not hand the call directly to the remote object but to the corresponding Virtual Processor (step 2). This is done by describing the call in a Job-object, that is passed using a Voyager-call to the Virtual Processor. The proxy holds a Voyager reference to the Virtual Processor and since Voyager implements call forwarding, this reference is also valid after migration. There the call is queued in the CallQueue (step 3) and at its turn transformed to a local call to the actual object (step 4). The results are first returned to the Virtual Processor and from there to the calling proxy (step 5). If the call returns references to other local objects (say of type `B`), for each of these a proxy is constructed and the proxy is handed back so that a local reference turns into a remote reference in the eyes of the client. The local object is registered to the Virtual Processor and the proxy is moved to the calling party.

For the construction of the compiler a tool called OpenJava [Tat97] has proved to be very helpful. OpenJava allows the extension of Java through a meta object description protocol. It has a very elaborate Java parser producing an object parse tree. The meta language allows to define changes in the grammar and to define operations on this parse tree. It then produces Java-code according to these definitions. Therefore, it was not necessary to construct the compiler from scratch, but allowed the development of Dejay as a Java "dialect".

3.10 Unresolved Problems

Some problems could so far not be resolved, mostly due to properties of Java. For compatibility purposes to regular Java, Dejay proxies are type compatible to the object they represent. This is done by inheriting from the original class. Since inheritance is not possible from classes defined as final, these classes can not be type compatible. Also some classes are not defined to be serializable (i.e. Enumeration) or have no class definition (i.e. arrays). For such classes a wrapper class has to be built.

To execute objects that are migrated their according class code is needed on the new platform. This has either to be in the local class path of the new machine or has to be downloaded from the net. Currently class definitions can be downloaded from a central source server, for example the machine that the main program is started on.

Some properties of Java clash with the idea of distribution. Java contains variables called static that are valid for a class and not an instance. Static variables in Dejay are only defined for a single Java Virtual Machine, but not globally. Also, variables with an according visibility can be accessed from other classes. Dejay does not support this but requires methods (get and set) for variables.

4 Examples

To demonstrate Dejay two small examples are given. The first is a "Hello World" example and is intended to show how remote creation and remote access to objects is using Dejay. The second is the dining philosophers example, that shows that the same implementation can be used to express concurrency as well as distribution.

4.1 Hello World

Consider the following class `HelloWorld`. It is stored in a file `HelloWorld.dj` and is a legal Dejay class file:

```
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 `dejayc` 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.

4.2 Dining Philosopher

As further example to demonstrate Dejay, we use Dikstra's well known dining philosophers. This example does not detect or avoid deadlock. Deadlock must be avoided by usual means (for example by having one philosopher pick up the fork in opposite order). It is intended to show the unification of distribution and concurrency using migration of Virtual Processors.

Five philosophers sit around a table set with five forks and a dish of spaghetti. Each Philosopher needs two forks to eat. If he is done eating he puts down the two forks and thinks until he gets hungry again and tries to pick up two forks once again. Since Dejay is designed for distributed systems, we assume that all of the five Philosophers as well as the five forks could potentially be placed on different machines in a distributed system. Each Philosopher thus has to retain a remote reference to a fork on his left and on his right.

```
import java.io.Serializable;
import dejay.base.*;
```

```
public class Philosopher implements Serializable {
   DjFork left_fork;
   DjFork right_fork;
   String name;
   DjButler james;


public Philosopher (String myName, DjFork left, DjFork right, DjButler butler) {
      left_fork = left;
      right_fork = right;
      name = myName;
      james = butler;
      System.out.println("Philosopher "+name+" created");
   }

   public void dine() {
      while (james.isServing()) {
         eat();
         think();
      }
   }

   private void eat() {
      while(hungry()) {
         // use both forks in parallel, asynchronous call
         private DjForkResult left_done;

left_done = (DjForkResult) left_fork.use( Dejay.ASYNC, new DejayResult());
         DjForkResult right_done;

right_done = (DjForkResult) right_fork.use( Dejay.ASYNC, new DejayResult());
         // now block until both calls return
         System.out.println(name +" is eating");
         Dejay.wait(left_done);
         Dejay.wait(right_done);
      }
   }


private void think() {System.out.println(name +" is thinking...");}

   private boolean hungry() {
      // random algorithm to determine hunger
   }
}
```

To set up the proper scenario, a butler should lay the table and seat the philosophers. The butler can create the forks on one (or different) remote Virtual Processors. Since he needs the forks as parameters for the creation of the philosophers, he does this synchronously. He can than seat, i.e. create, the philosophers, each in a separate Virtual Processor. Since he calls the creation asynchronously, he can create all of them in parallel. Finally he serves the meal, i.e. calls the dine-method on each one, again in parallel. In this case the asynchronous call is mandatory since the dine method will loop forever. The according code in Dejay is presented below; for simplicity the example is restricted to two philosophers:

```
public class Butler {

   DjProcessor processor1, processor2, processor0;
   DjFork fork1, fork2;
   DjPhilosopher jan, thorsten;
   boolean serve = true;
   DjButler proxy;

   public Butler() {
      // X, Y, Z. are IP addresses and port of Voyager demons
      // These can be different or identical so that
      // the application could run on one or many machines
      try {
         processor0 = new DjProcessor(X);
         processor1 = new DjProcessor(Y);
         processor2 = new DjProcessor(Z);
      } catch (ConstructProcessorFailedException e) {
         System.out.println(e);
      }

      // Lay Forks on table
      fork1= new DjFork(processor0);
      fork2= new DjFork(processor0);

      // Seat philosophers and give each two forks

jan = new DjPhilosopher("jan", fork1, fork2, processor1, new DjButler(this));
thorsten = new DjPhilosopher("thorsten", fork2, fork1, processor2, new DjButler(this))

        System.out.println("Table is set up");
        jan.dine(Dejay.ASYNC);
        thorsten.dine(Dejay.ASYNC);
        System.out.println("Meal is served");
    }


    public boolean isServing(){
        return serve;
    }
```

Each of the philosophers or the butler could now be moved from one machine to another, simply by calling the `moveTo()` method on either the object or its Virtual Processor. The following are two methods that change the physical placement. The first gathers all philosophers and forks on one machine, the other spreads them out again in an altered order. Note that such a movement does not stop the philosophers meal, all remote references remain valid and the philosophers keep using the correct forks.

```
    public void assemble() {
        // interrupt the meal
        serve=false;
        // move all philosophers to one machine X
        jan.moveTo(processor0);
        thorsten.moveTo(processor0);
        // continue the meal
        serve=true;
        jan.dine();
        thorsten.dine();
    }

    public void distribute() {
        // interrupt the meal
        serve=false;
        // spread philosophers and forks on different machines
        jan.moveTo(processor2);
        thorsten.moveTo(processor1);
        // continue the meal
        serve=true;
        jan.dine();
        thorsten.dine();
    }
```

This program can be started in the usual way by calling a `main()` method. This automatically creates the initial Virtual Processor in which the butler resides.

```
      public static void main(String args[]) {
         Butler james = new Butler();
         ...
         james.assemble();
         ...
         james.distribute();
      }
   } // class Butler
```

5 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. However the current runtime behavior is very promising. We have implemented the distributed calculation of Mandelbrot-sets using different communication techniques like Sockets, RMI and Voyager and compared it to an implementation using Dejay. The results are shown in figure 10. The figure shows results for the full picture, as well as for zooms into communication intensive (red picture) and computation intensive areas (black picture). The usage of Dejay introduces some additional overhead, however this seems to be acceptable for a prototype and can be further reduced. The advantage is that the calculation server implemented in Dejay can remotely be instantiated and migrated at runtime.

Also some additional services like binding and grouping are being developed. Binding to existing objects is possible through a name service. An object can be bound to a name and located by others using this name and its location. This can be implemented using RMI, CORBA or Voyager mechanisms and has been tested but a final integration into the language has not yet been done. It would even be possible to smoothly integrate an infrastructure like Jini to discover such objects.

Grouping of virtual processors on a yet higher level of abstraction is currently being investigated. Grouping is not difficult to incorporate but it has so far not been decided whether this grouping should be transitive, reflexive, and/or hierarchical or not. Such a mechanism could be used to dynamically reconfigure attachments. The currently most favored approach is to introduce an extra object `ProcessorGroup`, similar to `ThreadGroup`, that can hierarchically combine Virtual Processors that will migrate (or persist) together.

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 and since Voyager offers a mobile agent mechanism, a Virtual Processor can also be seen as a mobile agent. This aspect is currently being evaluated.

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 for 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.

6 Related Work

It has repeatedly been discussed that Java, as is, is not very well suited for distribution [BLS97]. Especially the current mechanism for remote method invocation has widely been criticized [PZ96]. 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).

The most closely related projects are Voyager and JavaParty. Voyager [Obj98] has already been mentioned and some of its weaknesses were pointed out. Nevertheless, Voyager is a great product and the Dejay project has profited much by relying on it. JavaParty [PZ96] is similar in that also a Java dialect is being developed. It extends Java by adding an extra keyword remote and provides a new compiler. Its main goal is to provide an improved RMI but its focus is on multiprocessor machines and tightly coupled networks.

Many other projects exist that examine distribution or concurrency concerns for object oriented languages. Emerald [BHJ$^+$87] had a strong influence, for example on DOWL [Ach93] or Beta [SB93]. Here, migration of groups of objects is expressed by explicit attachment. Its relation to Dejay is discussed in [Bog98]. For a recent overview on concurrent OO-languages see [BGL97].

7 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

[Ach93] B. Achauer. The dowl distributed object-oriented language. *Communications of the ACM*, 12(9), September 1993.

[BGL97] J. Brioit, 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.

[BHJ+87] 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.

[BLS97] 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.

[Bog98] Marko Boger. Migrating objects in electronic commerce applications. In *Proceedings of Trends in Distributed Systems for Electronic Commerce*, 1998.

[COS98] COSMOS. Electronic contracting with cosmos, 1998. www.ponton-hamburg.de/cosmos/.

[Frø96] Svend Frølund. *Coordinating Distributed Objects. An Actor-Based Approach to Synchronization.* MIT Press, 1996.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, second edition, 1997.

[MY93] Satoshi Masuoka and Akinori Yonezawa. *Research Directions in Concurrent Object-Oriented Programming*, chapter Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. MIT Press, 1993.

[Obj98] ObjectSpace. Objectspace, 1998. www.objectspace.com.

[PZ96] Michael Philippsen and Matthias Zenger. Javaparty - transparent remote objects in java. *Concurrency: Practice and Experience*, 9(11), November 1996.

[SB93] 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.

[Tat97] M. Tatsubori. Openjava, 1997. www.softlab.is.tsukuba.ac.jp/~mich/openjava/.

	central	distributed
single party access	1	2
multiple party access	3	4

Table 1: Different implementation alternatives for negotiation in COSMOS.

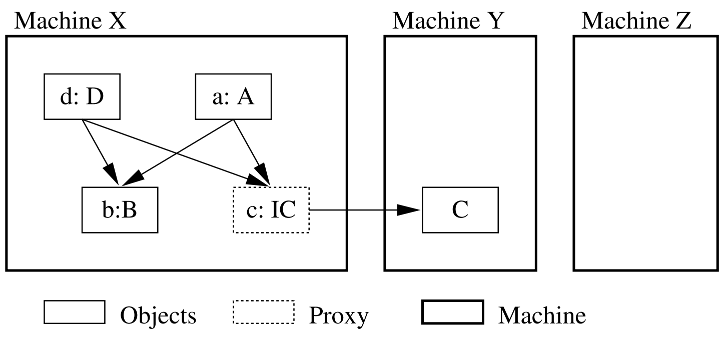

Figure 1: Different Reference types in Voyager.

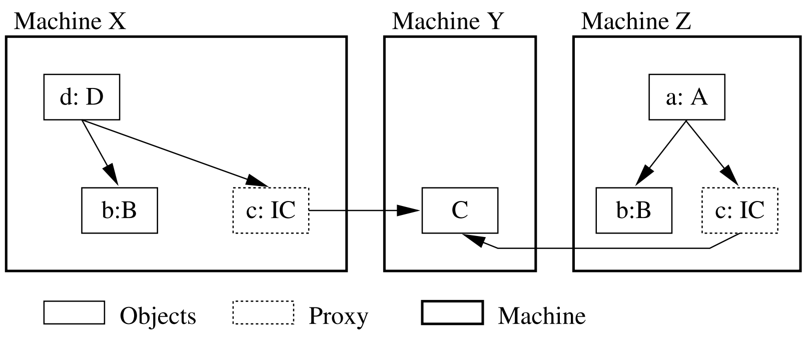

Figure 2: After moving a, two copies of b exist, causing potential inconsistencies.

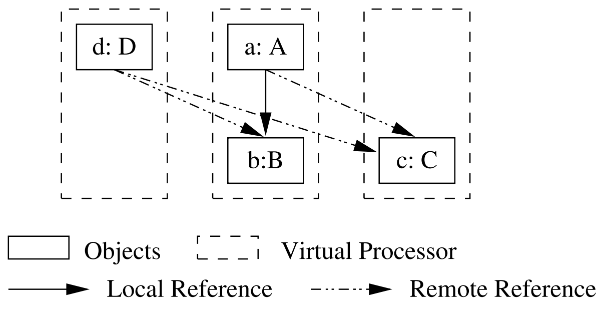

Figure 3: Virtual Processors containing objects.

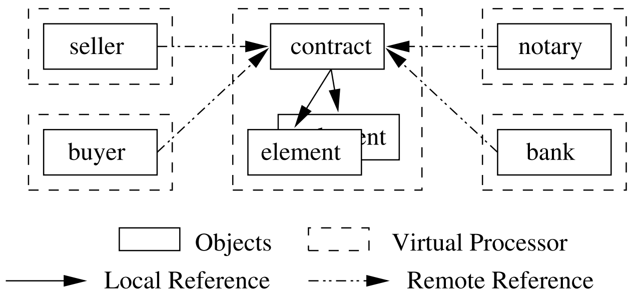

Figure 4: Multiple user access to a contract.

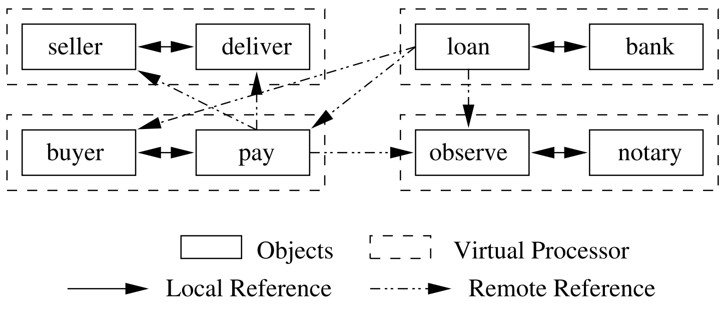

Figure 5: Distributed Workflow.

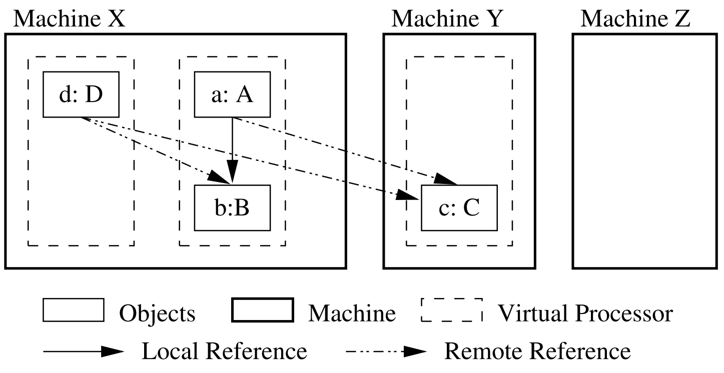

Figure 6: Objects and References in Dejay.

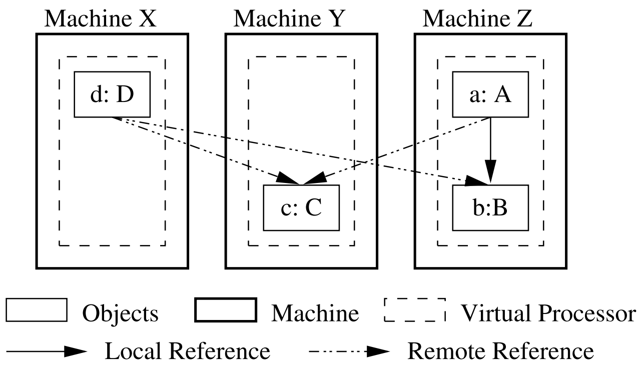

Figure 7: After moving a, one copy of b exists. No inconsistencies.

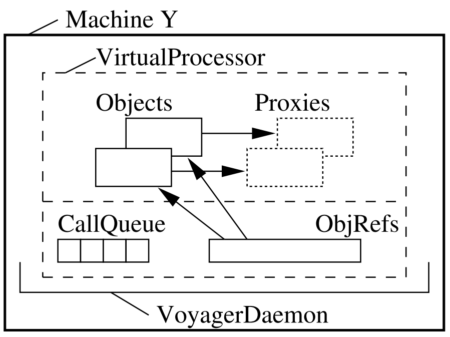

Figure 8: Implementation structure of a Processor.

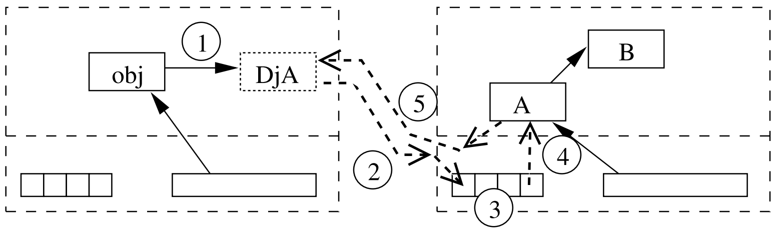

Figure 9: Implementation of a remote method call.

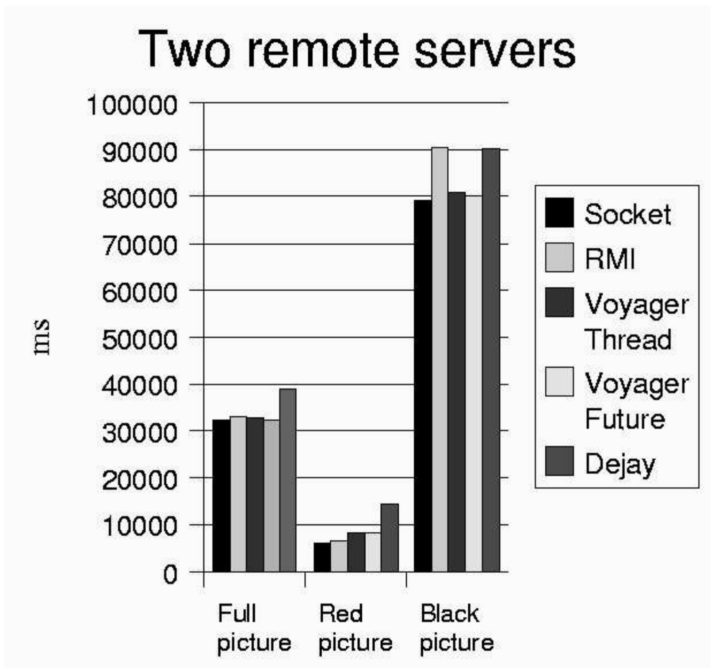

Figure 10: Distributed calculation of Mandelbrot-sets with different communication mechanisms and for different areas.