Migrating Objects in Electronic Commerce Applications

Marko Boger
Hamburg University - Department of Computer Science- Distributed Systems Group
Vogt-Kölln-Strasse 30, 22527 Hamburg
boger @ informatik.uni-hamburg.de
Phone: ++49 +40-5494 2343 Fax: ++49 +40-5494 2328

Abstract

Electronic Commerce is a field of application that is distributed by nature where different parties share information and work concurrently and cooperatively on objects, potentially distributed over a large scale network like the internet. In such an environment client/server architectures reach the limit of their capability. Non-centralized distributed architectures with object and code migration are more suitable. This paper presents a distributed extension to Java named Dejay which is designed to simplify the design and development of such distributed systems. Concurrency and distribution are expressed using the same mechanism, virtual processors. These processors represent one thread of control. They contain groups of objects and manage their synchronization and migration over distributed networks. It is used as implementation language for distributed electronic commerce applications.

Keywords: Java, Distribution, Migration, Concurrency, Electronic Commerce

1 Introduction

Electronic Commerce is an evolving application field for distributed systems where industry and research still investigate for an appropriate implementation platform. The requirements are immense: security aspects, fault tolerance, runtime problems, heterogeneity of hard-, middle and software are just a few problems that need to be dealt with. This makes the search for a suitable implementation platform very difficult.

The two single most often cited technologies in this context today are Java and CORBA. Both are well suited for client/server architectures but have shortcomings in distributed applications that require object or code migration. In CORBA this is simply not possible. Java offers techniques for code migration. Object migration is possible but very inconvenient. Some extensions to Java like JavaParty [Philippsen 97] or Voyager [ObjectSpace 97] exist that ameliorate this but can also not deliver a sufficient solution. One of the main unsolved problems is the grouping of objects that are to be migrated together.

This paper first investigates the needs in electronic commerce and other distributed applications that are not met by current technologies, the migration of objects. Different concepts for object migration, fine grained object migration and virtual objects, are discussed and their shortcomings pointed out. As a new approach, the

paper presents an extension to Java called Dejay. It incorporates migration and concurrency with the mechanism of virtual processors. This mechanism is a very expressive yet simple way to group objects and migrate such groups as whole. It also allows modeling of concurrency. It replaces and simplifies the thread mechanism given by Java.

2 The need for migration of objects

Today's programming paradigm for internet or intranets and applications like electronic commerce on such nets is client/server. Java as well as CORBA lend themselves to this paradigm. But many applications exist that are distributed by nature. In these applications it does not suffice to hand over remote references and transmit data of simple data types as in CORBA or copy objects as in Java RMI. What is missing is the ability to migrate objects or groups of objects. Such a mechanism would ease or even enable the development of distributed applications. It would increase usability as well as designability of such systems. Examples come from many application areas like robot-systems, computer aided manufacturing, process control, distributed simulations or workflow systems.

An example in the field of electronic commerce is an electronic contract system. An electronic contract is the electronic version of a usual business contract. A contract involves different parties that need access to it at different places and times. The advantages of an electronic contract are at least threefold. First, for such a contract electronic media can be used. It could simply be sent via internet from one party to another. But it could also contain a program that could help filling out forms or check the validity of entered data. Second, it can be selfpresenting. While a written document has only one view, such an electronic contract can have different views, revealing differently important information to different parties. Third, it can be active. The contract itself can control the forthcoming of its execution. It could control deadlines and give according messages or warnings. It could be specification as well as controller of a workflow.

These examples would benefit greatly from the ability to move objects or groups of objects from one site to another. The contract for example is composed of different components of objects. These objects and their code need to move to where it is needed, i.e. to present itself or to check the forthcoming of a workflow. If desired only one component needs to be moved, for example only the component presenting a delivery address to the computer of an expediter. This is where current client/server systems fail. With CORBA it is not possible to move objects. In the case of CORBA the introduction of object mobility is conceptually extremely difficult, since CORBA systems allow different implementation languages on different sites, making different object representations unusable. The OMG is working on suggestions to solve this problem but so far with little success. Java allows for simple object movements but does not provide a simple solution and is not sufficient for complex systems. Using RMI the movement of an object requires an object factory of appropriate type and the passing of all relevant state information to create a copy of the original object on a different site. Nevertheless Java allows the migration of code to a different site at

runtime and offers good network communication facilities, what makes Java a good starting point to build a system that supports object migration.

2.1 Mobile Objects

In object-oriented systems the most obvious subject of migration is the object itself. Several research projects have addressed object migration, the most frequent approach being to keep the distribution as transparent as possible and only moving objects where explicitly requested. The most known and one of the first projects following this concept is Emerald, developed at the University of Washington [Emerald 87], [Jul 89]. Emerald was specifically designed for the needs in a network of autonomous computers connected by a local area network in a homogeneous environment. Objects in Emerald are referenced throughout the distributed system; a call to a remote reference is a remote method invocation. This makes the network transparent to the programmer if this is desired. But to achieve an efficient implementation objects can be moved to the place they are referenced from and can then answer method calls on the same machine. An object can be moved from one machine to another at any time, even while one or more methods are being executed.

This mechanism has attracted a lot of attention. It is widely agreed that it demonstrates the general suitability of the object-oriented approach to distributed systems. It has been adopted in several different object-oriented languages like Trellis/DOWL [Achauer 93] and Beta [Brandt 94]. Also in Java several projects are adopting the approach of a fine granularity, making it possible to move objects of an arbitrarily small (or big) size. Examples of this are JavaParty [Philippsen 97] or Voyager [ObjectSpace 97].

The problem that arises using fine grained distribution is the following: when moving an object, what other objects should also be moved? Objects communicate with other objects. Moving one object to a machine where it is often accessed to reduce network communication can result in an eventually larger network communication if the moved object needs to communicate to objects that where not moved. Emerald proposes to build groups of objects that are to be moved together by attaching objects to each other. When declaring an attribute of a class, this attribute can be marked by the keyword *attached.* This attachment is recursive and transitive but not symmetric.

The design of attachment relations is a tedious and error-prone work. The code needs to be changed explicitly , spoiling distribution transparency and the ease of design and maintenance. Emerald and its successors have shown that distribution transparency is feasible. But they have also shown that in order to achieve efficient implementations object migration is necessary and that objects need to migrate in groups. Nonetheless, they can not provide a sufficient solution to grouping.

2.2 Virtual Objects

In usual distributed languages the call to a remote object is transparent, as in Java's RMI and in CORBA. A project proposing to drop this distribution transparency is Voyager from ObjectSpace [ObjectSpace 97]. Voyager is an ambitioned project that

aims at setting a new standard for distributed programming, integrating as well as replacing other techniques like RMI, CORBA and Agents. It is based on and completely written in Java. It offers a compiler that can automatically prepare any class in Java source or byte code for distributed computing. Besides propositions for autonomous migration, persistence, security, CORBA-integration and multicasting, Voyager introduces a different approach for referencing and migrating remote objects.

In Voyager the difference of a call to a local and to a remote object is made explicit. Any Java class can be compiled by a Voyager compiler to produce what Voyager calls a virtual class. For example a class C would be compiled to the virtual class VC. This virtual class implements a superset of the interface of the original class; an instance of this class, the virtual object, serves as a local representative of a remote object of class C. This is very similar to proxies or to stubs used in RMI or CORBA, but while proxies or stubs are hidden from the view of the programmer, in Voyager these virtual objects are explicitly used. A remote object is never called directly. Instead its local representative, the virtual object is called, that in its turn handles the remote call (an instance c of the virtual class VC of the remote enabled class C as shown in Figure 1). This means giving up distribution transparency without changing the syntax of a method call. Yet a reference to a remote object is different to a local object since its type is of a virtual class.

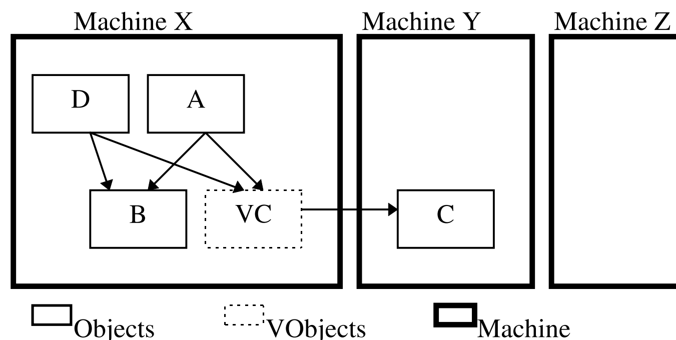

Figure 1: Different Reference types in Voyager

Voyager supports object migration by adding a *moveTo* method to each virtual object that, when called with an IP address or host name as argument, will move the referenced object to the specified machine. Assume we have an object *a* of class A on machine X as in Figure 1. This object can be moved to machine Z by sending it the method call *a.moveTo(Z).* The question is what happens to references and referenced objects. A virtual object can simply be copied to machine Z without affecting the rest of the system. If object *a* has a reference to a normal object, say *b* of class B, then *b* needs to be copied too, so that the reference is still correct. But if *a* changes the state of *b* then this change can not be seen on the original copy of *b* (see Figure 2). In order to avoid this problem all references from an object that is to be moved need to be of virtual type. If this rule is not obeyed inconsistencies will acurre. Concluding it can be noted that Voyager does offer object migration and if used carefully at a fine grained level. But it offers no grouping mechanism (like attachment in Emerald). It is the

programmers responsibility to avoid problems caused by migration. This way the development of distributed systems remains a difficult and error-prone work.

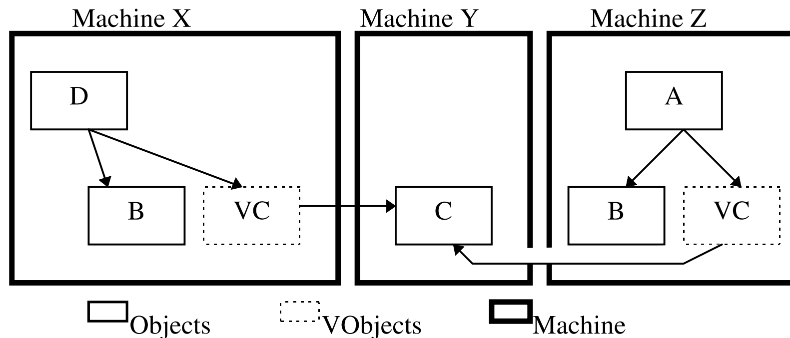

Figure 2: After moving *a*, two copies of *b* exist. Inconsistencies possible!

2.3 Virtual Processors

If the object is not the proper subject of migration since its granularity is too small, then what is the proper subject of migration. An interesting approach is the migration of a thread like in [Mathiske 96]. I consider this as too close to hardware concepts and would like to present a similar but more abstract concept, the concept of virtual processors.

A virtual processor is an autonomous thread of control capable of supporting the sequential execution of instructions on one ore more objects. This is an abstraction of the concepts of a physical processor, heavy weight processes and light weight processes, often called threads. It can be implemented by either of these and the objects that are executing within a virtual processor should not be aware of the form of implementation used. Every object is assigned to exactly one virtual processor but a virtual processor can contain several objects. Objects with tight couplings can be assigned to one processor, making this concept a grouping mechanism for objects. This concept is introduced by [Meyer 97] and is currently being implemented as an extension to the programming language Eiffel. Meyer shows that this concepts integrates well with object-orientation, synchronization and inheritance. In his approach objects are automatically assigned to a virtual processor by a runtime mechanism. Similar to the concept of virtual objects in Voyager, it is differentiated between local and remote references, here by introducing a new keyword *separate* to the language.

Meyer does not explicitly use this mechanism for object migration but primarily for expressing concurrency. The language he uses, Eiffel, has so far not been ready for code migration making migration of virtual processors troublesome. Nevertheless, this mechanism is well suited for migration if the underlying system supports code migration, like Java, and object migration, like Voyager. I propose to incorporate this mechanism to Java and Voyager and to extend this notion to migration, as explained in the following section.

3 Dejay - A mechanism for Distribution in Java

Dejay is an extension to Java, aiming at simplifying the programming of distributed systems. It is part of a research project on electronic commerce and is intended as implementation platform for distributed electronic commerce applications such as distributed and collaborative contracting tools and other applications. It is especially intended for problems that are distributed by nature and run in an environment where communication is sufficiently reliable and fast but too costly to be neglected. Such environments are the internet, extra- and intranets as well as local area networks.

The syntax of Dejay is very similar to that of Java. In fact, Dejay is a subset of Java; the threading mechanism of Java is completely replaced and all keywords concerning threads are not allowed in Dejay. A compiler translates Dejay to Java code. Therefore Dejay is compatible with Java; all existing Java classes and objects can be called and used in Dejay. Also Dejay classes and objects can easily be integrated from other Java classes.

The mechanism that replaces Java threads is that of virtual processors. It is used to model concurrency and to group and migrate objects at a same time. By this I achieve a considerable simplification of my language compared to Java and provide a simple and secure migration mechanism. Every object is created in and controlled by exactly on processor. If only one thread of control is needed the use of processors remains completely implicit. If several treads of control are needed a processor is created for each one. This can be done either on the same machine or on several different machines connected by a network. Processors can be moved at runtime and all objects contained within it are automatically moved with it. Objects can reference other objects in the same processor as in usual Java. They can also reference and use objects in different processors independent of its location. But references to objects in different processors are marked. For this the mechanism of Voyager, virtual classes, is used. A reference to an object outside of its own processor has to be of virtual type, which is checked by the Dejay compiler.

3.1 Creation

A processor is a Voyager object and can be created on any reachable remote machine running Voyager. To create it remotely the constructor of a virtual processor is passed the name of the intended machine.

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

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

In Dejay each object belongs to and is managed by a processor. Objects can be called from outside the processor by using their virtual objects. But the processor executes calls in a sequential manner. This greatly simplifies synchronization. No synchronization is needed between objets contained in the same processor since there is only one thread of control. Synchronization between objects in different processors is much simpler than in Java. Objects are always used exclusively. Using high level

constructs for synchronization allows the efficient supported by the compiler. 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 synchronisation mechanisms like semaphores by high level mechanisms simplifies the language and allows automatic generation of efficient code. Different mechanisms for expressing synchronization constraints are currently being investigated, including that of Eiffel [Meyer 97] using pre- and postconditions as wait conditions, of Java using an extended *synchronized* keyword, separate synchronization specifications or synchronizers as discussed in [Frolund 97].

An object can be instantiated in the usual Java fashion. In this case it belongs to the same virtual processor as the object calling the constructor. Objects can also be created on a different virtual processor. In this case a local representative is created in a similar way as in Voyager. This local representative is passed the reference to a remote processor and will instanciate an object on the processor specified. To differentiate between a local and a remote reference, remote references have to be of a virtual type. Within class A we can write

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

// create an object on processor Y
VC c = new VC(Y);
c.some_method();
```

which will result in the situation shown in Figure 3. On machine X another processor and object of class D with some virtual references is added.

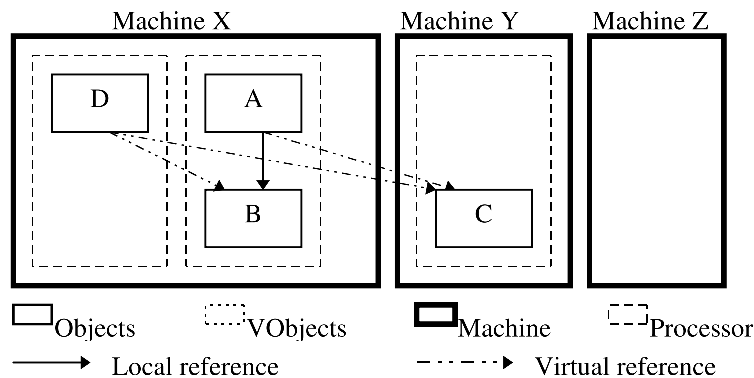

Figure 3: Objects and References in Dejay

3.2 Migration

A virtual processor can be moved from one machine to another simply by calling a *moveTo* method. As argument this method accepts an IP address, a host name or a reference to another virtual processor. It then migrates 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. Sending an object contained in a processor the *moveTo* message will also result in the movement of the entire processor and all its contained objects.

```
// move processor p1 to machine Z
p1.moveTo(Z);

//equivalent: move a to Z
a.moveTo(Z)
```

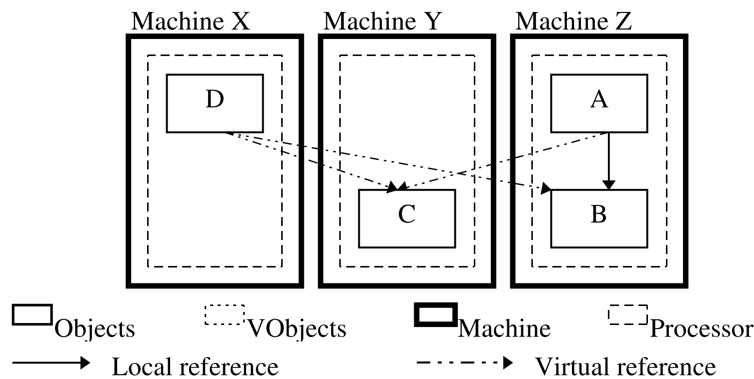

Figure 4: After moving *a* one copy of *b* exists. No inconsistencies.

In this way no inconsistencies can accure. No extra copies like in Voyager are needed. All references remain valuable. Migration becomes simple and secure. Objects are always moved as group, keeping related objects together. Communication between objects belonging to different processors is the same, whether the processes reside on the same or on different machines. But moving the processors to the same machine can reduce communication costs by an order of magnitude.

3.3 Concurrency

The mechanism of processors is also used to express concurrency. Concurrency can be used to perform actions in parallel. This 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 actively waiting for input while others continue. In Java these are two distinct concepts. The first requires communication via sockets or RMI, while the second is that of threads. In Dejay they are unified to one concept. If two processes run on the same machine, execution is not parallel but only simulates it. If one of them is moved to a different machine it turns to real parallelism.

In Dejay method calls can be synchronous or asynchronous. In Java method calls can only be synchronous. Calling a method on a remote object, i.e. using RMI, will result in long waiting times since the calling object will block until the result is returned. To simulate asynchrony 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 to develop. But a simple mechanism for asynchronous calls is vital for distributed

programming. Only using asynchrony true parallelism on different machines can be achieved. Therefore, Dejay incorporates and facilitates the use of asynchronous calls. It relies on the mechanism of Voyager but extends it to further ease the use of asynchrony.

In Voyager each call to a remote object is handled by an object called messenger. Voyager offers different types of messengers. The default is a synchronous messenger that returns a result when the call to a remote object is completed. It does not appear explicitly in the code and requires no further preparations. It has the look and feel of normal Java method calls. To make asynchronous calls an asynchronous messenger object is passed as additional parameter to the call. The call returns immediately with a reference to the messenger. The actual result of the call can be looked up later by querying the messenger object.

Dejay extends this notion to a mechanism called wait-by-necessity and is similar to the mechanism discussed in [Meyer 97]. By default all calls to objects in different processors are asynchronous. This remains completely transparent to the programmer. But the thread of control continues directly after the call is set off. It stops and waits if the result of the call is actually used.

```
// create an object on processor Y and use it
VC c = new VC(Y);
result = c.some_method(); // asynchronous, continues immediately
// do other calculations
...
// use results
some_var = result.some_operation(); // blocks until result is delivered
```

This makes the use of concurrency transparent and simple. Yet, if the programmer needs to have direct control over the call mechanism, he can explicitly fall back on the mechanism of Voyager.

3.4 Design

Dejay is from the start tailored to support and simplify the design of distributed applications. A new modeling and design method and suitable models to express concurrency and distribution and a graphical tool are currently being developed hand in hand with Dejay.

In current analysis and design methods like OMT, Booch or UML, little attention is paid to concurrency and distribution. The general approach is to develop a class (or object) model that depicts the dependencies and relations of classes. These are inheritance, association and aggregation. Other models like scenarios, use cases, state models or data flow models help finding the methods and interfaces needed in the class model. Nonetheless, the development is centered around the class model. Unfortunately the class model can not express concurrency, since concurrency does not appear on class level. It appears on object level. Objects only exist at runtime and are therefore not an appropriate means to model concurrency at design time. But references to objects and method calls can be used. They are dynamically linked to objects at runtime, but their type is known at compile time. In Dejay the type of a reference to a remote object is clearly differentiated from a local reference. Also each

remote reference points to an object in a remote processor. Processors are used to express distribution as well as concurrency. Using Dejay, a model expressing concurrency and distribution can therefore be built upon these references. Our model is further described in [Boger 96].

3.5 Implementation

The implementation relies heavily on Voyager. The class *VProcessor* is produced by compiling the class *Processor* using the Voyager compiler. This *Processor* class maintains a queue of incoming calls and dispatches them one at a time, retaining other calls until the dispatched call is done and the result is returned, thereby insuring a sequential processing. It creates a new Voyager daemon which is completely under its control. Each time a call is dispatched from the queue the call is simply handed over to this Voyager daemon. The *Processor* is a relatively thin layer encapsulating the Voyager mechanism.

The *Processor* class is the only class directly compiled by the Voyager compiler. All other classes within a Dejay system are compiled by the Dejay compiler. For the construction of virtual classes it internally relies on the Voyager compiler but changes its output to redirect calls to the appropriate virtual processor.

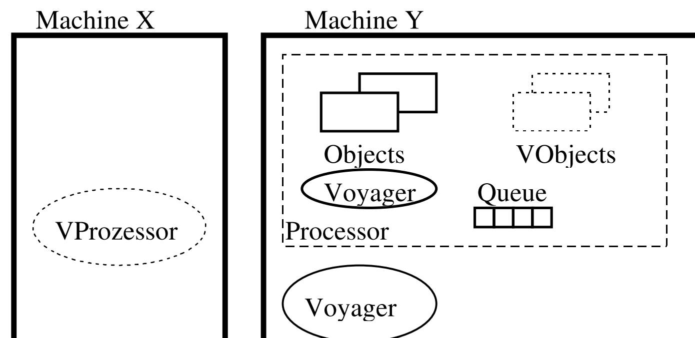

Figure 5: Implementation structure

The compiler of Java is constructed using OpenJava [Tatsubori 97]. It allows the extension of Java through a metaobject description protocol. This replaces the need for constructing a compiler from scratch and thus makes the development of a Java dialect relatively simple.

4 Related Work

It has repeatedly been discussed that Java, as is, is not very well suited for distribution [Philippsen 97],[Brose 97b]. Therefore there is a great interest in Java-based or Java-extending solutions that aim at improving the distribution abilities of Java.

JavaParty [Philippsen 97] improves the mechanism of RMI by slightly extending the Java language and providing a new compiler. Different to Dejay it is intended for fast local nets or massively parallel machines.

Interesting but also specialized to small and fast networks is the concept of virtual memory. A famous project in this area is Linda, where tuples of data can be written into and read from a globally accessible memory called tuplespace. Sun is working on an integration of the Linda approach to Java, called JavaSpace.

A couple of projects only use Java as platform for portability and implement a completely different language on top. Some of these are focused on functional programming [Hall 97] or scripting languages, like Ambit [Cardelli 97].

The descirbed system has very close relations to mobile agent systems, especially those based on Java like Mole, Aglets, Odyssey or Voyager [Cockayne 97]. Mobile agent systems are intended for autonomous movement of programs to environments that are insecure, unknown and not trusted. Dejay is not primarily designed for such environments but tries to simplify the design of closed distributed applications. It can therefore avoid a lot of the overhead that mobile agent systems have to deal with. Nevertheless, I believe that Dejay can be extended for such environments.

5 Summary

This world is a distributed world and real objects move in this real world. We model this world with software objects and we have long started to do this in a distributed environment. Many problems need to be solved to achieve this goal, including concurrency control, synchronization, movement of code, distributed resource allocation, distributed garbage collection, efficiency, security and others. Java is a well suited programming language to tackle this problem but has so far not succeeded in solving this problem. The paper pointed out that the granularity of movement is an important issue and presented different approaches. A fine grained approach where the subject of movement is the object itself like in Emerald is problematic, mainly because the grouping of objects becomes a tedious and error-prone work. The concept of virtual objects as followed in Voyager was discussed. It has many advantages but also does not solve the problem of grouping objects. An approach was discussed where the subject of movement is a virtual processor. This eases the grouping and management of objects and simplifies synchronization and concurrency.

I propose to combine the approach of virtual processors with the concept of virtual classes on top of Java. Dejay was presented, a new programming language extending Java, following this approach. Dejay is intended as a language for distributed applications in networks of computers. In Dejay grouping and migration of objects and concurrency are expressed using the same concept of virtual processes. This makes the design and development of distributed systems simpler as compared to using Java or Voyager. Dejay is part of ongoing projects in electronic commerce at Hamburg University and will be used as implementation language for distributed electronic commerce applications.

6 References

[Achauer 93] B. Achauer: „The DOWL Distributed Object-Oriented Language“. Communications of the ACM, Sept. 1993.

[Boger 96] M. Boger, H.W. Gellersen: „On Models in Object-Oriented Methods-Critique and a new Aproacht to Reversibility“. Technology of Object-Oriented Languages and Systems, TOOLS 19, Paris, Feb. 1996.

[Brand 94] S. Brandt, O. Lehrman Madsen: „Object-Oriented Distributed Programming in BETA“. Feb. 1994.

[Brose 97a] G. Brose: „JacORB: Implementation and Design of a Java ORB“. Proc. DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Aplications and Interoperable Systems, Cottbus, Germany, Chapman&Hall, Oktober 1997.

[Brose 97b] G. Brose, K.-P. Löhr, A. Spiegel: „Java does not Distribute“. Proc. TOOLS Pacific ´97, Melbourne, Australia, November 1997.

[Cardelli 97] L. Cardelli: „Ambit“. http://www.luca.demon.co.uk/Ambit/Ambit.html, 1997.

[Cockayne 97] W.R. Cockayne, M. Zyda: „Mobile Agents“. Manning Publications, Greenwich, 1997.

[Emerald 87] N. Hutchinson, R. Raj, A. Black, H. Levy, E. Jul: „The Emerald Programming Language Report“. Technical Report 87-10-07, University of Washington, Washington, 1987.

[Frolund 97] S. Frolund: „Coordinating Distributed Objects“. The MIT Press, Cambridge, Massachusetts, 1997.

[Hall 97] D.A. Hall: „Applying Mobile Code to Distributed Systems“. Doctoral dissertation, University of Cambridge, Cambridge, June 1997.

[Jul 89] E. Jul: „Object Mobility in a Distributed Object-Oriented System“. Doctoral dissertation, University of Washington, Washington, 1989.

[Mathiske 96] B. Mathiske, F. Matthes, J.W. Schmidt: „On Migrating Threads“. Journal of Intelligent Information Systems, 1996.

[Meyer 97] B. Meyer: „Object-Oriented Software Construction, 2nd. Ed.“. Prentice Hall, New Jersey, 1997.

[ObjectSpace 97] ObjectSpace: „Voyager Core Technology“. Http://www.objectspace.com/voyager/ 1997.

[Philippsen 97] M. Philippsen: „JavaParty - Transparent Remote Objects in Java“. Concurrency: Practice and Experience, November 1997.

[Tatsubori 97] M. Tatsubori 97: „OpenJava“. http://www.softlab.is.tsukuba.ac.jp/LANG/mich/openjava/, November 1997.