Ulrich Becker

Nauru: A concept for the management of object distribution

Technical Report TR-I4-02-01 ◆ April 2002

Department of Computer Sciences 4
Distributed Systems and Operating Systems

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Univ. Erlangen-Nürnberg • Informatik 4 • Martensstr. 1 • 91058 Erlangen • Germany
Phone: +49.9131.85.27277 • Fax: +49.9131.85.28732
E-Mail: i4@informatik.uni-erlangen.de • URL: http://www4.informatik.uni-erlangen.de

TECHNISCHE FAKULTÄT
(Faculty of Engineering Sciences)

Nauru: A concept for the management of object distribution

Ulrich Becker

ubecker@cs.fau.de
Informatik 4, University of Erlangen-Nürnberg
Martensstr. 1, D-91058 Erlangen, Germany

Abstract: Performance and reliability of a distributed application critically depend on where the application objects are placed. Current off-the-shelf middleware offers full control over the distribution, but urges the developer to scatter distribution-specific statements all over the source code.

Our concept, named Nauru, provides distribution control in connection with the application design. The developer expresses distribution requirements together with the UML model of the application. Such requirements can enforce to place objects together, and place restrictions on their location. By referring to the UML model, we can use associations and state models to specify those requirements. This facilitates a specification of distribution requirements that is declarative, compact and easy to understand.

To realize the distribution specification, we first preprocess the source code so that the relevant model information and the resulting distribution requirements are kept in the implementation. The actual distribution is determined dynamically by the Nauru runtime system.

1. Introduction

Inherently distributed applications, such as distributed workflow applications or groupware applications, become more and more important. These applications have rigid requirements regarding their reliability and performance. To fulfill these requirements, one has to carefully choose the distribution of the application objects.

When we examine how control over the distribution is supported by approaches to the development of distributed object-oriented applications, we identify two poles: At one pole, we have systems that provide full distribution transparency. These systems aim to turn arbitrary applications into distributed ones. The opposite pole consists of systems that mainly provide remote interaction between objects; with these systems, the developer has to deal with all distribution issues himself.

We found that neither of these poles is well suited for the development of the applications that we target: Approaches that offer full distribution transparency do not provide the required control of the distribution, whereas pure middleware approaches put too much burden on the developer.

Therefore, we have chosen a position between these poles for our concept, named Nauru[1]. The software developer specifies the distribution requirements in connection with the application design. Such requirements can express that objects must be placed together, and place restrictions on their locations. By referring to the UML model, we can use associations and state models to specify those requirements. This enables an easy-to-understand and compact specification of the distribution requirements. Furthermore, the distribution specification is completely separated from the source code of the application.

The realization of the distribution specification is done mostly automatically by two components, the *Nauru weaver* and the *Nauru runtime system*. The Nauru weaver is a preprocessor for the mostly distribution-unaware application source code. It weaves code that interacts with the middleware and with the Nauru runtime system into the application source code. When the application is executed, the Nauru runtime system dynamically determines a distribution that fulfills the distribution requirements.

The rest of this paper is structured as follows: In the next section, we examine the two opposing approaches to the development of distributed object-oriented applications. In Section 3, we present our concept, Nauru, and the components that are needed for its realization. Section 4 compares Nauru to related work. In Section 5, we discuss the trade-offs that have to be made to gain simplicity, and the feasibility of an implementation. In the last section we give our conclusions.

2. Transparency versus distribution control

When we examine approaches to the development of distributed object-oriented applications, we identify two poles: At one pole, we have systems that provide full distribution transparency. The opposite pole consists of systems that

1. Hence the name: The island Nauru lies at the equator, so that it is positioned between the poles.

mainly provide remote interaction between objects; with these systems, the developer has to deal with all distribution issues himself.

Most approaches clearly belong to one of these poles: Systems like *Pangaea* [14], *JavaParty* [12], *Coign* [5], or *Juggle* [13] aim to turn arbitrary applications into distributed ones, either by preprocessing the source code, by providing a special execution environment, or by a combination of these. On the other hand, most real world distributed applications are programmed directly on top of a middleware like *CORBA* [10], *Java RMI* [15], or *DCOM* [9]. We think that the best solution is not found at one of these poles, since both have substantial disadvantages:

While full distribution transparency eases development, it also means that the developer loses control over the distribution of the application. This is often a problem, because performance and reliability depend on where the application objects are placed within the distributed system. To achieve good performance, objects that interact heavily must be placed together. To obtain robustness, independent node failures have to be taken into account: If two objects on different nodes interact with each other, this can fail if one of the nodes crashes. A robust application has to detect such failures and recover from them in an application-specific way. To ease this task, it is desirable to control the distribution in a way that independent failures occur only at places where they can be conveniently handled.

Systems that provide full distribution transparency focus on performance optimization, either by dynamically adapting the distribution to the interaction patterns (e.g., Juggle), or by determining a distribution based on source code analysis or profiling results (e.g., Coign, Pangaea). Although a hand-optimized application may provide better performance, the performance will be sufficient for many applications.

Robustness is a harder problem: Recovery of failures requires deep knowledge of the application, and therefore cannot be automated. Since no control of the distribution is possible, failures have to be expected at every object interaction. To obtain robustness, the developer would therefore have to provide a lot of error recovery code to handle all these situations. Systems that provide full transparency usually ignore this problem: Node failures are considered unlikely, and therefore no error handling is done. If node failures occur, the application will have to quit, which is unacceptable for the considered application class.

Some applications are also affected by another problem: The underlying assumption of most of these systems is that all nodes are equivalent, so that objects can be placed everywhere without affecting the application's functionality. This is not true for objects that need to use features that are specific to certain nodes. For such applications, systems that provide full transparency are unsuitable.

At the other pole, the situation is inverse: The software developer has full control of the distribution, but this is achieved by burdening the developer with implementation details and by giving up transparency. Distribution control is exercised by substituting local object creation with object creation via a remote factory; if the middleware provides object migration, this is typically used by adding migration statements to the code. We call this approach *imperative distribution control*.

The most serious problem with this approach is that distribution control is scattered across the source code. As a result, there is no single place where the distribution can be examined and edited, and the implementation is unnecessarily mixed with distribution issues. These are the problems of cross cutting and code tangling that are well known from the literature about Aspect Oriented Programming (e.g., [8]).

3. Nauru

3.1 Characterization of the application class

Nauru is targeted at an application class for which distributed workflow applications or groupware applications are typical representatives. These applications are interactive, run for a long time and have rigid requirements regarding their performance and reliability. The objects of such applications can be broadly grouped into three categories ([6] , [7]):

- *Interface objects* implement the user interface of the applications. These objects are tied to the nodes on which the interface part of the application runs.
- *Entity objects* represent long-lived application information and contain parts of the business logic. They typically have no limitations regarding their distribution.
- *Control objects* are short-lived objects that implement the application's use cases. Their distribution requirements vary: depending on the interaction patterns of the use case, placement at the user interface or at a specific server may be required. This may even dynamically change depending on the phase of the use case execution.

3.2 Specification

Systems with imperative distribution control force the software developer to implement a concrete distribution. With Nauru, we have chosen a different approach: Our distribution specification language is geared towards the specification of distribution requirements, not how they are achieved.

The specification consists of a set of distribution statements, each of which expresses one distribution requirement. The distribution statements are based on the UML model of the application, so the specification can refer to associations and state diagrams. These model elements represent information about object relations and the phases in an object's lifecycle. As we will demonstrate below, this information is well suited for the specification in a compact, declarative way.

The distribution specification is represented as a separate XML file. By physically separating the distribution specification from the model and the implementation, we achieve a clean separation of the distribution aspect and the application logic. While we use XML for our examples below and for the internal representation, we do not intend this format to be directly used by the developer. Instead, we propose the integration of our concept into extendable modelling tools, such as TogetherJ or ArgoUML [1].

The specification language provides four distribution statements that are able to express the distribution requirements of our application class:

- *collocation*: With a collocation statement, objects that have to be placed on the same node are tied together.
- *require-property*: When objects have to be placed on nodes with certain properties, this can be declared with a require-property statement.
- *fix-object*: Objects that must not be migrated can be declared immobile with fix-object.
- *declare-replicable*: If an object can be safely replicated, this can be declared with the declare-replicable statement.

Due to the limited available space, we will only describe collocation and require-property in detail.

Application scenario

To demonstrate the use of these statements, we will present them within the context of our application scenario. Our application scenario is a distributed workflow application for the creation and maintenance of a news website. For the sake of brevity, we use a grossly simplified view of the application. The entity objects are articles, images and layouts. Images and layout objects belong to a single article object and are connected to it through a composition relation (Fig. 3.1 a). The article and its auxiliary objects are used by authors and designers while the article is created and edited. Before the article can be published on the Web server, it has to be converted into a format that is suitable for publication, such as HTML. This is done by a HTML generator object. The generator stores the result in a web page object that can then be passed to a Web server for publication (Fig. 3.1 b).

In our scenario, we distinguish between several node types for different purposes: Workstation nodes are used for the interactive creation of content. Database nodes are used for persistent storage of entity objects. They also offer high availability, so that they are well suited for the placement of entity objects while they are not used otherwise. Nodes with high computing power are used to generate web pages and for the Web servers.

Collocation

Conceptually, collocation is a binary object relation that ties two objects together, so that they are always placed on the same node. In the distribution specification, we use associations from the UML model to describe collocation requirements: A collocation statement refers to an association from the UML model to specify which objects should get collocated. Whenever two objects are related via this association, they are placed together. The association is referenced by an identifier that uniquely identifies the association within the model. The identifier is part of the model and is automatically assigned when the association is created, and does not change thereafter.

Example: The article forms a logical unit with its images and the layout information, and most use cases involve this unit as a whole. Therefore, an article and the objects that belong to it should be placed together to improve performance and reliability. The following distribution statement shows how this can be achieved for the article and its images:

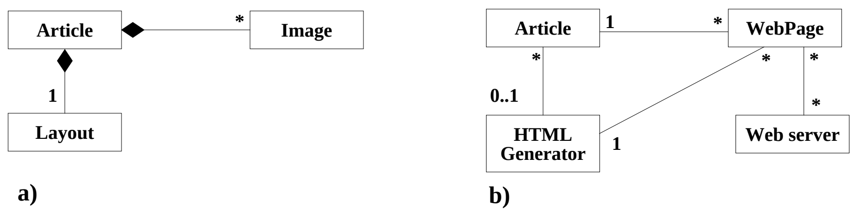


Fig. 3.1 Static application model

```
<collocate>
   <association-id>article_image_01</association-id>
</collocate>
```

Because the UML model is used as the basis for distribution specification, the explicit information about object relations that is present in the model can be directly used to easily express collocation requirements. With approaches that are based on the implementation, this is significantly harder, because information about object relations is only implicitly present in the implementation.

With the collocation specification described above, changes in the distribution occur only in conjunction with the creation or deletion of object relations. To specify distribution requirements that depend on the dynamic state of an object, we use UML state diagrams (Fig. 3.2). They describe what abstract states an object has, and how state changes are triggered by events. Our use of state diagrams for the specification of dynamic distribution requirements occurs through *state restrictions*. They can be added to distribution statements and limit the validity of the specification to certain abstract states of the participating objects.

Example: While a HTML generator creates a web page, both objects interact intensively and should be collocated. The relation between a generator and its web page can be used to specify the collocation, but the collocation would be too long-lived: The relation persists during the whole lifetime of the web page, whereas collocation is only required while the collocation is generated. With state restrictions, this can be achieved by limiting the collocation to the phase where the web page is generated:

```
<collocate>
   <association-id>webpage_generator_01</association-id>
   <state-restriction role=”WebPage”>
      <state>UnderConstruction</state>
   </state-restriction>
</collocate>
```

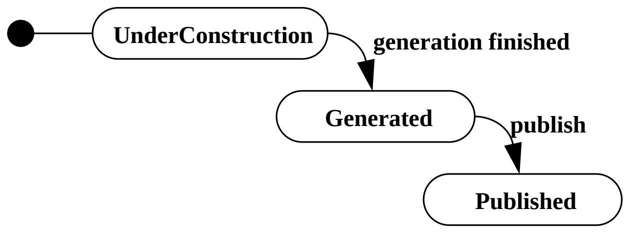


Fig. 3.2 State Diagram of Article

Required properties

Normally, objects can be placed on arbitrary nodes. If an object requires a special kind of node, such as a compute server, the developer can specify this with the statement *required-property*. To do so, the developer first defines properties that constitute his terminology for the specification of placement requirements. These properties are then used to specify placement requirements for certain objects.

Example: A HTML generator should be placed on a fast server. To achieve this, the software developer first defines a boolean property *IsComputeServer*. This property can then be used to specify that generator objects should be placed on a compute server:

```
<require-property>
   <type>HTMLGeneratorr</type>
   <property>
      <name>IsComputeServer</name>
      <value>true</value>
      <operator>exactly</operator>
   </property>
</require-property>
```

When the application is executed, the Nauru runtime system dynamically chooses appropriate nodes for the placement of generator objects. Section 3.4 gives more information on how this is done.

Placement preferences that change dynamically can be specified with state restrictions in the same way as this is done with collocations.

3.3 Verification

The abstract specification of distribution requirements gives rise to a problem: It is possible that a distribution specification cannot be realized. This problem stems from the combination of collocations and placement requirements. Since collocation is a transitive relation, potentially large groups of objects have to be collocated at runtime. We call a group of objects that has to be collocated an *object cluster*. An object cluster may contain two kinds of conflicts that might be overlooked by the developer:

- Objects within an object cluster have conflicting placement requirements, so that the distribution specification is intrinsically inconsistent. For example, if one objects requires the Windows operating system and another object requires Linux, these objects can never be placed on the same node and thus violate the collocation requirement.

- For a certain distributed system, there is no node that fulfills the requirements of all objects in an object cluster. In this case, the conflict is specific to a distributed system and not intrinsic to the distribution specification. For example, if one object requires the Windows operating system and another objects requires JDK version 1.3, this is not an intrinsic conflict. Whether this requirement can be fulfilled can be answered only with respect to a specific distributed system.

If a conflict occurs at runtime, the consequences can be fatal. Because the software developer relies on the fulfillment of the distribution specification, a violation of the specification can lead to unhandled remote exceptions, or can result in objects that do not work properly because they are placed on an unsuitable node.

To prevent such situations, we are developing a verifier that is able to detect such conflicts in advance. The verifier is used at two points in the development process (Fig. 3.3): To detect intrinsic conflicts during the development, and to detect conflicts specific to a distributed system when the application is deployed.

An algorithm for the detection of conflicts has one mandatory and one desirable property: it must detect every conflict that can occur at runtime, and it should not detect false conflicts that cannot occur in reality. Our algorithm guarantees the first property, but may detect false conflicts in certain situations. If the conflict is possible according to the model and the distribution specification, but cannot occur due to implicit constraints, this cannot be detected by our algorithm. In that case, model and distribution specification have to be modified so that verification is possible.

3.4 Realization

The support for the automatic realization of the distribution specification is divided between two components that are deployed at different phases in the application lifecycle (Fig. 3.3): The Nauru weaver is used prior to the compilation to weave distribution specific code into the application source code. The Nauru runtime system is deployed during the application execution; it dynamically distributes the application objects so that the distribution specification is fulfilled, and provides an abstraction layer for the middleware.

Nauru weaver

The basis for the weaver are the distribution specification, the application sources, and the underlying UML model. The application sources are mostly

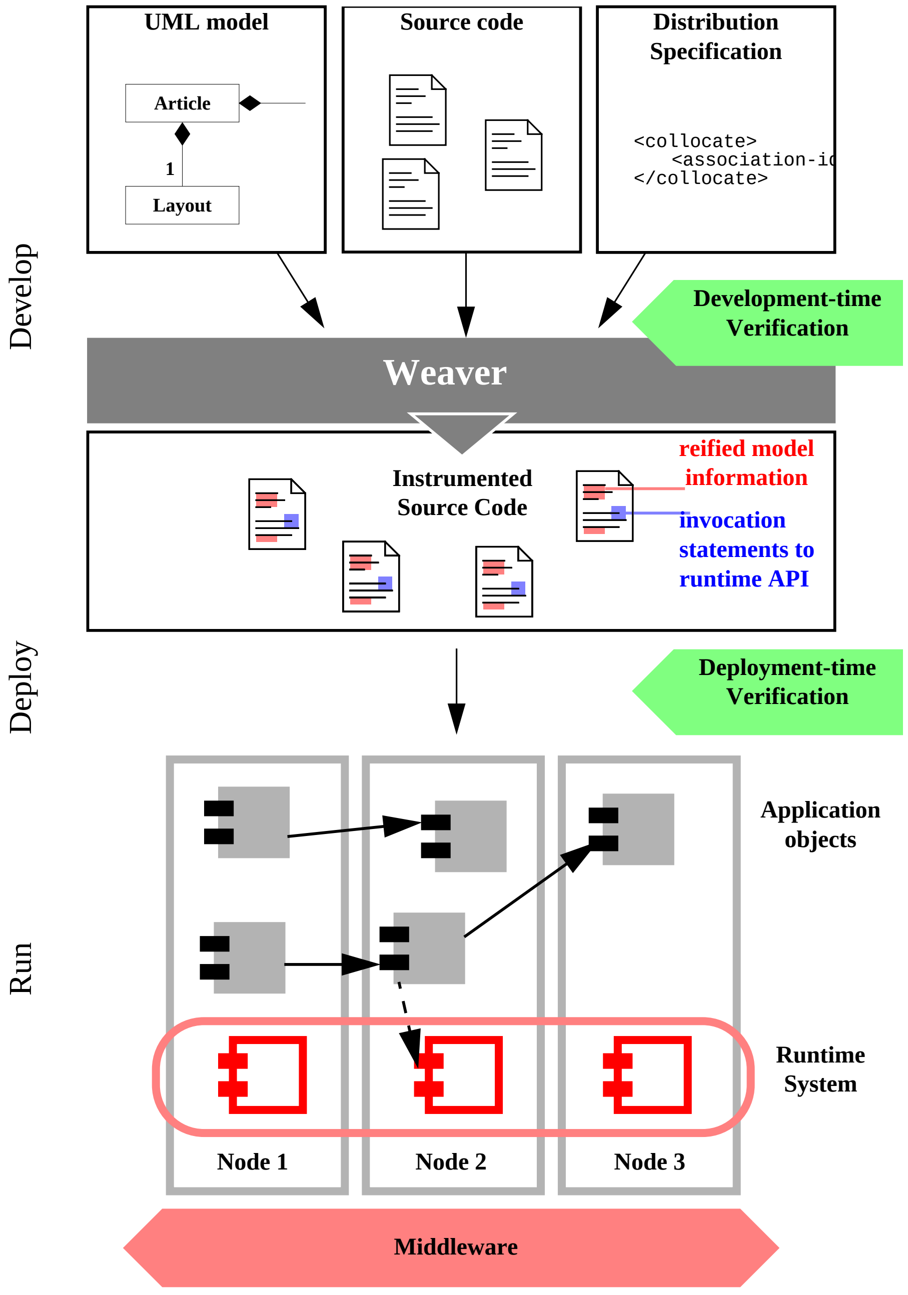


Fig. 3.3 Nauru Architecture

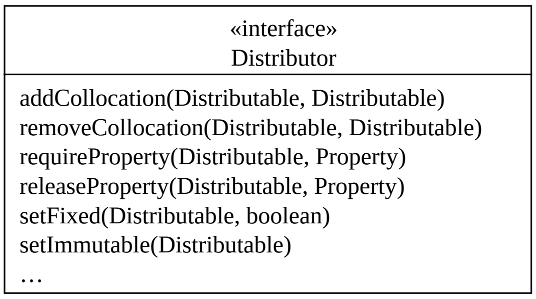
unaware of distribution issues. Besides exception handlers for distribution related failures, there is no code that differs from a non-distributed implementation. To prepare the source code for a distributed environment, the weaver adapts and extends the source code in various ways:

- Because the distribution specification is based on elements of the UML model, some model information has to be retained at runtime. The weaver first determines a mapping between the relevant model elements and the implementation. For example, for an association that is used as the basis for a collocation, the weaver determines what instance variables or container objects are used to implement this association. It then weaves in code that intercepts the control flow whenever these data structures are modified, and passes control to a handler operation.
- This handler operation is also generated by the weaver. Its purpose is to keep track of the model state that corresponds to the implementation state, and to notify the runtime system whenever a relevant change takes place at the model level.

The weaver also checks whether the software developer handles remote exceptions where necessary. If this is not the case, the weaver notifies the software developer and quits.

Nauru runtime system

The Nauru runtime system dynamically decides where to place the application objects according to the distribution requirements, and uses the underlying middleware to migrate the objects as necessary. To be able to make these decisions, the runtime system needs a complete and up-to-date view of the application's distribution requirements. This view is acquired through the Distributor interface that is accessible to every application object (Fig. 3.4):



«interface»
Distributor

addCollocation(Distributable, Distributable)
removeCollocation(Distributable, Distributable)
requireProperty(Distributable, Property)
releaseProperty(Distributable, Property)
setFixed(Distributable, boolean)
setImmutable(Distributable)
…

Fig. 3.4 Notification interface for changed distribution requirements

The operations of the API correspond to the distribution statements; e.g., the operation *addCollocation()* is invoked to notify the runtime system of a new collocation requirement between two objects. The code that is generated by the weaver uses this API to transform the events into changes in the required distribution; its direct use by the programmer is not intended.

Whenever the runtime system is informed of a change in the distribution requirements, it checks whether the current distribution still fulfills the requirements. If not, it determines a new distribution. Usually there is more than one distribution that fulfills the requirements; the runtime system then has to choose between these distributions. To do this, it uses a cost function that currently only takes the migration costs into account: The runtime system chooses the distribution that requires the smallest number of object migrations.

After the new distribution is determined, the underlying middleware is advised to perform the necessary object migrations. We require a middleware that provides *strong object migration*. With strong migration, an object can be migrated with its execution state, so that the migration can occur at any time and the execution is continued at the target node. With weak migration, only the data is migrated, and operation executions that were active at the time of migration are completed at the old location. To guarantee consistency, the object must not migrate until all these executions are completed. If migration is triggered asynchronously, the application continues its execution before the object migrates, and the application state becomes inconsistent with the distribution. If this happens, the distribution requirements are not fulfilled, which must never happen. On the other hand, if migration is triggered synchronously, this can lead to long delays and possibly even to deadlocks. Therefore, strong migration is a prerequisite for our concept.

While the runtime system is conceptually one instance for the whole application, in reality every node has its own instance of the runtime system. Each instance keeps the information about the distribution requirements of its local objects. Distributing this information over the nodes has several advantages:

- By placing distribution information on the nodes where the corresponding objects reside, the runtime system can handle many requests locally. Only changes that involve objects on more than one node require remote interactions between runtime systems.
- There is no central instance that could become a bottleneck.

Due to the division of responsibilities between weaver and runtime system, the runtime system can remain fully generic: Knowledge about the application model and the distribution specification is only needed for the mapping from application events to changes in distribution requirements for individual objects. This mapping is performed by code that is generated by the weaver. The interface of the runtime system is then used by this code to inform the runtime system of the changes.

To be able to consider placement requirements, the runtime system needs to know what properties are defined, and what properties are provided by the nodes that participate in the program execution. Currently, this information is provided in a textual configuration file that has to be provided in the deployment phase.

4. Related Work

Emerald [2] provides a programming language and a runtime system for the development of distributed applications. This system was the first to offer fine-grained object mobility, together with language support to control it. Emerald provides language constructs to migrate objects, to fix objects at a certain node, and to attach objects to other objects. While at first glance, these constructs show some similarities to the distribution statements of Nauru, a closer examination shows significant differences: In Nauru, every distribution statement is declarative, and the developer can rely upon the fulfillment of the distribution specification. In contrast, Emerald's move command can be used to place two objects together, but another move command that is invoked by another thread can cancel this effect. Similarly, because attachment is not symmetric, a migration of the attached object to another node cancels the effect of the attachment. The reason for these semantics is that Emerald's constructs for migration control are mainly geared towards performance improvement. Therefore, they do not provide the same support for the development of robust applications as Nauru. Furthermore, because mobility control is achieved through language constructs, Emerald necessarily falls into the category of imperative distribution control. As a result, Emerald also suffers from the problem that mobility control is scattered over the application source code.

FarGo [4] aims to enable the development of efficient and reliable distributed applications. Like Nauru, FarGo seeks a middle course between control over the application distribution on one side and separation of application logic and distribution issues on the other side. Unlike Nauru, FarGo gives up the uniform object model: At the syntactic level, FarGo distinguishes between fine-grained language objects and coarse-grained *complets* that are the basic unit of distribution. Internally, a complet consists of language objects; one of them is called the anchor object that provides the interface to the complet as a whole. The key to distribution control in FarGo are the references between complets: FarGo provides several kinds of *relocation semantics* that can be associated with a complete reference. When a complet that references another complet migrates, the relocation semantics of the reference determines how the referenced complet is treated. Some of the options that are available are to leave the referenced complet where it is, to also migrate the referenced complet, and to replicate the referenced complet. In contrast to Nauru, FarGo allows the dynamic change of relocation semantics at runtime. While this enables the dynamic adaptation of the distribution, we think that this is a problem due to the differences between the relocation semantics: For example, whether a complet is replicated during the application execution has to be considered in the application logic. Therefore, we do not think that arbitrary changes to the relocation semantics should be allowed. Furthermore, we consider it problematic to give up the uniform object model, since this restricts the reusability and maintainability of the code.

5. Discussion

In the preceding sections, we outlined how the distribution of an application can be specified by using a small set of simple distribution statements. This raises the question what trade-offs have to be made to gain this simplicity, and whether this approach is realizable.

If we compare the expressiveness of the Nauru specification language to imperative distribution control, it is clear that the latter is more expressive: Imperative distribution control imposes no limitations on how and when the distribution changes, because the developer can place the respective distribution statements at arbitrary places in the code. In Nauru, the distribution depends on the model state of the application, so that the distribution can only change in conjunction with object relations or abstract states. We think that Nauru is sufficiently expressive for a large class of applications, and that the benefits of simplicity outweigh the reduced expressiveness. To achieve the effect of a single Nauru distribution statement in the imperative model, several statements at different places in the code are required. Furthermore, it is very hard to determine from the code how the distribution will be, and whether it leads to the desired performance and robustness. This is much simpler in Nauru: Because the distribution specification is specified separately, there is a single place where the distribution can be examined. Furthermore, a declarative specification as in Nauru is easier to analyze than the imperative model.

A characteristic of our approach is that in certain cases, the UML model has to be adapted in order to specify and verify the desired distribution. We do not consider this a problem, because we do not regard distribution as orthogonal to the application's design. To achieve robustness and good performance, distribution issues must be taken into account at early stages of the application development, and therefore naturally influence the design.

An issue that we have not discussed so far is the increased reliability that can be achieved by replicating critical objects and placing them on different nodes. We do not support this because it would require a dislocation statement to enforce the placement on different nodes. Such a distribution statement makes it impossible to statically check the distribution specification for conflicts. Furthermore, we do not think that replicas of the same logical application object should be handled as distinct application objects. Instead, we propose to hide replication behind a single logical object. This can be achieved by middleware that supports fault-tolerant objects, such as the AspectIX [3] middleware.

When it comes to the realization of Nauru, the major challenge is to bridge the gap between the model and the implementation. Consistency between model and implementation is a prerequisite for this, but due to the round-trip engineering capabilities of recent modelling tools, this is no longer a problem. Still, the weaver needs to know the mapping between model elements and the implementation. Our prototype uses naming conventions to identify instance variables and operations that implement associations and events. If these heuristics fail, the developer can provide an explicit mapping. Due to the modular architecture of our weaver, the determination of the mapping can be adapted quite easily. With more and more code that is generated automatically by case tools, it is possible to develop tool-specific weaver modules that know how the tool generates code, and can thus reliably determine the mapping.

An implementation issue is our requirement for strong migration. As outlined in Section 3.4, strong migration is crucial for our approach. Because no off-the-shelf middleware offers strong migration, our runtime system works with a simulator instead of a really distributed middleware. Nevertheless, we expect strong migration to become available in off-the-shelf middleware.

6. Conclusion

We presented Nauru, a novel approach to the development of distributed applications. Nauru gives the developer full control over the application's distribution, so that good performance and reliability are possible. At the same time, the developer is burdened with far less difficulties and inconveniences than with imperative distribution control. The key to Nauru's simplicity is the small set of easy-to-understand, but expressive distribution statements that are used to specify the application's distribution. A main cause for both simplicity and expressiveness is that the distribution specification is based on the UML model of the application. By doing so, we can use the information about object relations and object lifecycles that is captured in class diagrams and state diagrams. This information has shown to be a good basis for a compact, declarative and comprehensible distribution specification. In contrast to imperative distribution control, the developer can focus on what the distribution requirements are, not how they have to be realized. The realization of the distribution specification is done mostly automatically by the Nauru weaver and runtime system.

At this time, we have prototype implementations of the weaver and the runtime system that show that an implementation of our approach is feasible. An implementation of the verifier is currently under development. To get a fully working prototype, these partial prototypes have to be integrated with each other and with a middleware that provides strong migration.

7. References

1. ArgoUML Homepage: http://argouml.tigris.org/
2. A. Black, N. Hutchinson, E. Jul, H. Levy, *Fine-Grained Mobility in the Emerald System*. ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988
3. Franz J. Hauck, Erich Meier, Ulrich Becker, Martin Geier, Uwe Rastofer, Martin Steckermeier: *A middleware architecture for scalable, QoS-aware and self-organizing global services*. In Proc. of the 3rd IFIP/GI Int. Conf. on Trends towards a Universal Service Market - USM, LNCS 1890, Springer, 2000; pp. 214-229
4. Ophir Holder, Israel Ben-Shaul, Hovav Gazit: *Dynamic Layout of Distributed Applications in FarGo*. Proceedings of the 1999 International Conference on Software Engineering, ACM Press 1999; pp. 163-173
5. Galen C. Hunt, Michael L. Scott: *The Coign Automatic Distributed Partitioning System*. Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI '99), pp. 187-200
6. Ivar Jacobson, *Object-oriented software engineering - a use case driven approach*, ACM Press, 1992
7. Ivar Jacobson, Grady Booch, James Rumbaugh, *The unified software development process*, Addison-Wesley, 1999
8. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopez, Jean-Marc Loingtier, John Irwin: *Aspect-Oriented Programming*. Proceedings of the European Conference on Object-Oriented Programming 1997 (ECOOP), LNCS 1241, Springer, June 1997

9. Microsoft, *Distributed Component Object Model Protocol*. http://www.microsoft.com/com/resources/specs.asp
10. Object Management Group, *Corba 2.6 Specification.* http://www.omg.org/technology/documents/formal/corba_iiop.htm
11. Object Management Group, *UML 1.4 Specification.* http://www.omg.org/technology/documents/formal/uml.htm
12. Michael Philippsen, Matthias Zenger: *JavaParty - transparent remote-objects in Java*. Concurrency: Practice and Experience, 9(11):1225-1242, November 1997
13. Michael Schröder, Franz J. Hauck: *Juggle - Eine verteilte virtuelle Maschine für Java*. JIT1999
14. Andre Spiegel: *Pangaea - An Automatic Distribution Front-End for Java*. Proceedings of the IPPS/SPDP Workshops, LNCS 1586, Springer, 1999; p. 93-99
15. Sun Microsystems: *Java Remote Method Invocation Specification*, JDK 1.3. http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmiTOC.html