Journal of Universal Computer Science, vol. 8, no. 1 (2002), 97-116
submitted: 21/12/01, accepted: 16/1/02, appeared: 28/1/02

On Second Generation Distributed Component Systems

Klaus Schmaranz
(Institute for Information Processing and Computer Supported New Media
(IICM), Graz, Austria
kschmar@iicm.edu)

Abstract: Two of today's most used buzz-words in the context of software development are the terms *Componentware* and *Distributed Object-System*. The combination of both ideas is then called a *Distributed Component-System*, meaning a componentware approach where the components are distributed across the network. Today's approaches fulfill the application developers' needs only partly. Also, most are more or less cumbersome to use. I want to call such part-solutions like e.g. Corba, Enterprise Java-Beans and others *first generation distributed component systems*. In fact, Corba has a different origin, but for the moment let me consider it to be a first generation componentware system, too.

In this paper I want to identify the requirements that have to be fulfilled to design and implement a second generation distributed component system. There is one main aspect behind all of the ideas of second generation systems: *a good distributed component system is one that the application programmers don't notice.*

The open-source project *Dinopolis* is currently in its early implementation phase and can be considered the first second-generation distributed component system according to the requirements that are identified in the following. Therefore the very basic cornerstones of Dinopolis are discussed at the end of this paper.

Key Words: Distributed Object System, Distributed Component System, Componentware, Java, Network Transparency Aspects, Robust Globally Unique Handles, Distributed Relations, Middleware, Dinopolis.

Category: D.1.5, D.2, D.2.6, D.2.7

1 Introduction

Ages ago (in terms of fast-evolving computer-science) the object-oriented software development paradigm was introduced. It allowed easy mastering of huge software packages by proper encapsulation. The OO-paradigm is still the paradigm of choice for good reasons for most modern programming languages. Properly applied (and only then!) the OO-paradigm makes, amongst other benefits, code-reuse easy, thus shortening turnaround times in the software-development cycle. However, due to the existence of OO programming languages, the term object-orientation is understood as an implementation of the encapsulation principle on the programming-language level today.

It did not take long until software developers wanted more than to reuse code by linking additional libraries to their software at compile-time and recompile the whole project. What was really desired was to have reusable entities with well defined interfaces that could be utilized during runtime. Nowadays these entities are called components. Software utilizing this paradigm is called *componentware*. The logical next step was to build software using the component-based approach with components that are distributed across a network, resulting in so-called *distributed component systems*.

Very soon it was recognized that some sort of standardized framework is needed which embeds the single components and adds the network-distribution facilities. Corba was one of the first approaches in this direction (see [OMG]). With the introduction of Java several other *distributed component frameworks* evolved, such as *ObjectSpace Voyager* (see [Voyager]) or Sun's approach called *Enterprise Java Beans* (see [EJB]). Let me call such systems *first generation distributed component systems* to indicate two important points:

- The existing frameworks are great and are definitively a large step into the right direction.
- Unfortunately all those systems also have some shortcomings creating a definitive need for what I would like to call *second generation distributed component systems*.

I do not want to start a religious discussion with the conclusion that all existing systems are bad, have to be discarded and something brand-new has to be invented. In my opinion this would result in "just another system" which overcomes identified problems but causes others, starting the next religious discussion. The scientists and developers who thought about and built the first-generation systems knew very well what they were doing. However, with every single problem that is solved, new ideas and new needs evolve.

Therefore, before I come to the analysis of what in my opinion really is essential for a distributed component system to be a *second generation system* I want to come straight to one of the conclusions of the following analysis: *it is possible and desirable to build a middleware-layer on top of existing systems. The architecture proposed in this paper creates a second-generation system as a combination of existing software with some additional mechanisms.*

Why this middleware-layer is desirable, how it works and especially what defines a second generation system in my eyes, is the topic of this paper. Therefore let me start at the very beginning with the identification of the requirements from the point of view of developers who write massively distributed applications, data- and computing-wise, in large scale heterogeneous networks.

2 Transparency – the Key to Distribution

There is a good reason why I started the introducion of this paper by mentioning the OO-paradigm: no matter if we are talking about classes, objects, modules or components – application developers want to utilize well-defined objects with well-defined interfaces in their programs. Whatever happens behind the scenes, whether or not there are components or distribution, heterogeneous networks or stand-alone systems, details have to be encapsulated and therefore hidden from the developer who works on a different abstraction level! The abstraction level for my considerations at the moment is: *everything is encapsulated in an object and developers do not care what the object encapsulates*. The common term for this kind of hiding complexity by encapsulation is *transparency*. It means to let "something" happen behind the scenes without putting the burden on the developers to distinguish between different situations.

I used the term *something* above, because there are many important transparency-aspects when considering distributed component systems. This will be discussed below.

2.1 Component Transparency – Part 1

This paper should be about *component*-systems and now I am at the level of discussing OO-aspects – why? The reason is that according to the definition above, the term *component* describes a reusable entity with well-defined interfaces that can be utilized during runtime. It does not have to be linked to a program statically during compile-time. Considering this definition even a whole application could be a component, as long as it provides a well-defined interface that makes it possible to access it from within a different application during runtime.

The goal is it to hide different kinds of components behind the scenes by encapsulating them. Inside applications the developers always work with objects, no matter if they are simple statically linked objects or if they encapsulate dynamically instantiable and accessible components.

At a first glance the requirement for component transparency is fulfilled by first generation systems. However, this indeed is only true at a first glance if we consider what happens behind the scenes when using e.g. Enterprise Java-Beans or Corba:

- Objects inside the so-called client represent external components: this is what we want.
- However, first generation systems usually do not give you control over the instances of external components. Usually clients cannot require that a new remote component shall be instantiated. They can just connect to an already instantiated component. Depending on the system it is also possible that components are instantiated on the fly when the first request occurs (e.g. with RMI's object-activation mechanism, see also [RMI]). Nevertheless all following requests from all different clients then refer to the same component.
- The fact that one can connect to instantiated components but not influence instantiation by means of system-immanent mechanisms leads to situations

where special “managing components” have to be written that take over this task.

Distributed component systems have to be more than just allowing to invoke methods on remote objects. Inside applications a request to create a new instance of a class is usual (e.g. in Java `my_object = new MyClass();`). Requiring component transparency an instance of `MyClass` could also require that a remote object is instantiated. In this case a remote instance has to be created and encapsulated by a local *stub*. The required *instance transparency* mechanisms also have to provide control over access-*exclusiveness*.

2.2 Network Transparency

Network transparency means that application developers do not have to know whether they are working with a local or with a remote-object. In order not to have to distinguish between different programming languages such as e.g. C++ which has pointers and Java which does not, the term *object-reference* will be used from now on. Object-reference means that an object is somehow held in an application. Application-programmers can work with this object exactly in the way as is pre-determined by the programming-language used.

Speaking in OO-terms we are talking about *classes* that define data-types and *objects* that represent instances of classes. A network-transparent object is an object that is an instance of a certain class, no matter if the instance is residing inside the application or on a different computer. It can very well be that several instances of a certain class are residing on the local machine and several other instances of the same class are residing on different remote machines. Nevertheless these instances all look the same to the application programmers: it is always possible to work with them as if they were residing inside their own application. The network-aspect is hidden behind the scenes. Therefore local and remote-objects are fully interchangeable.

The aspect of network transparency is one of the aspects that all first generation systems fulfill more or less in one way or another. Usually, some sort of static *stub-skeleton* model or a dynamic derivative of it is implemented to achieve this behaviour.

There is one problem remaining with network transparent objects, no matter how they are implemented: it is always possible that network connections fail. Therefore, no matter how well-designed and well-implemented the network-logic may be, some risk remains that requests fail due to network problems. Whatever topics may be discussed here, e.g. what happens to time-critical applications, the problem is system-immanent and has to be dealt with for every single application. Nevertheless it is easier to deal with this system-immanent problem if possible failure is a well-defined part of all object-interfaces. Fortunately the developers of the first generation systems seem to share this opinion and all systems provide more or less intelligent error-alert facilities for this case.

No matter how well network transparency is implemented in existing first generation systems, there are several aspects that are usually overlooked: having a remote object-reference in hands and working with it is one topic. The other topic is: how do we obtain such a remote reference? The following transparency requirements deal with exactly this problem more in detail.

2.3 Component Transparency – Part 2

To be able to find satisfactory answers to questions about obtaining remote references and dealing with them, it is necessary to pick up the *component transparency* thread again and find a detailed and comprehensive definition of the term *component*.

All first generation systems have one feature in common: they only deal with "their" native kind of components. First generation systems are not able to work with arbitrary content that is stored "somewhere" and consider this arbitrary content a component. Neither do first generation systems usually deal with other component systems in a cooperative manner in the sense that it would be possible to "combine" different systems.

Apparently, designers of first generation component systems did not take an approach that is general enough. I will now present a more holistic definition for distributed component systems:

- Looking at the world from inside a distributed component system, everything in the whole distributed world will be termed a component, no matter how granular components become. In this definition it does not matter whether a component represents a simple file in a filesystem (as a wrapper), a real remote object, a database-entry or maybe a remote object of a different system (e.g. Corba), or an application. It can also be that a component represents a document, e.g. an XML-document. This document is itself structured as a DOM-tree using components as nodes.
- Speaking of components that are composed of several sub-components two different cases can be distinguished:
  1. The entity that is represented by the component exists as one single piece and sub-components represent its logical structure.
  2. The entity that is represented by the component is made up of different chunks. Sub-components represent the individual chunks and the component that can be seen is in fact a container combining several sub-components to one logical virtual composite.
- A component has to be addressable in a globally unique way. This requirement applies to simple components (e.g. components wrapping single files)

and also to compound components. When addressing a compound component only the top-level component (i.e. the container) is addressed and the rest (i.e. where the parts come from) is hidden behind the scenes. This behaviour could be called *compound transparancy*. However, we must keep in mind that globally unique addressing in a dynamic world is a topic of its own (objects can move!). Aspects of globally unique dynamic addressing will be discussed in section 2.7.

– The world does not consist of arbitrary components hanging around somewhere in a vacuum being accessible just if we know the right key. It has to be possible to navigate through the componentspace either by means of directed and bidirectional relations and also by means of search-operations.
– Allowing relations between components means that arbitrary components can be interconnected, no matter in which system they reside. A detailed discussion about relations is postponed till section 2.8.
– Additional information like the content-type of the data that is encapsulated in an object or administrative data like author, creation date, etc. is also something that has to be handled in a transparent manner. Hence part of the component transparency requirement is unified handling of meta-data.
– Components have special services that they provide. Considering e.g. Java Beans, the Beans can be asked for their abilities. In our case components can also be wrappers for everything from simple content to applications, components can be made-up of sub-components, etc. A necessary requirement is that the abilities of the wrapped resource are passed on transparently to the components' users, i.e. the application programmers.
– There are cases where components can become active themselves, e.g. a component wrapping a timeplanner application must be able to pass-on triggers to other components in the system for reminders that come from the application.

After the above two rounds of discussion about aspects of component transparency it should be clear where transparency is needed. At a first glance it thus looks as if we could come to an exact definition of the term component now. However, some questions still remain if we look at requirements like globally unique stable addressing, relation-management or compound-components. Therefore let us first consider the remaining transparency aspects before presenting the final result.

2.4 Persistence Transparency

As mentioned, components can themselves be composed of several sub-components that can reside on different systems. For example it can happen that documents and meta-data describing the documents reside in different systems. It can happen that documents are stored in a filesystem, whereas additional meta-data such as keywords, descriptions, etc. are stored in a database. This happens e.g. very often in electronic publishing applications.

From the application programmers' point of view it is desirable to have one component that encapsulates the existence of different locations of the component's persistent data, making it unnoticable for users. This becomes especially important if a storage-system is replaced by a different one. As an example it can happen that meta-data is first stored in the filesystem and later, as the amount of data increases, all meta-data is moved to a database.

For this reason not only persistence transparency in the sense of static transparency is required. Persistence transparency has to cover the dynamic case too, where parts of the persistent state of a component can be moved to a different location. One more dynamic case would be that e.g. the persistent state of a component is stored as one single XML file in the filesystem including content and meta-data. In this case the application works with a simple component wrapping it. Later, the decision is made to move the meta-data to a database to make it searchable. Hence a simple component is converted into a combined component with meta-data from the server and the "rest" from the filesystem.

Back to the question in section 2.2: *how do we obtain a reference?* The requirement for dynamic persistence transparency rules out the use of addresses like URLs. The transparency requirements discussed later in section 2.5, section 2.6, section 2.7, section 2.8 and section 2.9 back up this conclusion.

2.5 Protocol Transparency

As has already been mentioned – components can reside "anywhere" and can move around. Cases like *first the persistent data of a component was in the filesystem on computer A and now the component is residing on an http-Server on computer B* and arbitrary many other scenarios are thinkable.

For this reason a protocol as a part of an address for a remote reference, like in URLs (see also [Berners-Lee et al 1994]), is unusable. The protocol to access a remote server, be it just a data-provider like an http-Server or a distributed object system like Corba, has to be completely encapsulated.

2.6 Schema Transparency

More or less the same problem, just from a different point of view, can be found when having a closer look at address-schemata (see [Terry 1984] and [Znati, Molka 1992]):

In most of today's systems addresses are somehow structured hierarchically, following an implicit or explicit schema. For example, file-systems have a directory-hierarchy that is used for two different purposes: *addressing* and *navigation*. The implicit schema here is the existence of hierarchical subdirectories that form a fully qualified path for addressing data. The explicit schema here is the way users or administrators structure the subdirectories to allow easy navigation.

Data in databases is accessed by queries and the queries are also based on a distinct schema that is designed by the developers. This schema is reflected in the queries needed to access data. In any case database-access and filesystem-access are completely different, even if we would encapsulate the protocol transparently as required in section 2.5. Who does not know the cryptic URL-encoded queries for accessing databases with a Web-front-end?

Imagine further that data-chunks are moved from one system to another (e.g. from the filesystem to a database or from one database to a different one with a different underlying schema). In this case all addresses obtained from the "old" system are unusable. Therefore it is also necessary to hide the schema from the developers.

It is worth to have another look at *navigation* in the address-space: Mixing up addressing and navigation is definitively a very bad idea, because every attempt to restructure the component-space would break the schema. Therefore those two issues, addressing and navigation, have to be strictly separated as will be pointed out in section 2.8.

2.7 Location Transparency

Several times it has been stated in this paper that moving components around can break the addressing mechanism. Hence the problems that can arise should be clear enough by now.

Let us summarize the resulting very strong requirement here: *remote handles have to be robust against all restructuring operations*.

These operations include moving components around, splitting them up into sub-components that are virtually merged in a container, merging split-up sub-components to one "real" component rather than a virtual container and moving sub-components around without breaking the virtually merged components.

From this requirement it finally becomes clear that addresses in the form of *pointers* are a problem, no matter if we take URLs or any other mechanism that points to a *location*.

The solution is what can be called a *globally unique handle*, which represents a symbolic name. The mechanism behind these handles is a little more complicated than it initially looks. There are several aspects of scalability which have to be considered. A naive lookup-service implementation would not work for a world-wide distributed componentspace. However, for our further considerations it is enough to know that in principle *globally unique handles* solve the location transparency problem. A detailed discussion how the scalability problems can be overcome is beyond the scope of this paper. These essential problems are solved and several algorithms have been developed to keep scalability very well under control (see [Schmaranz 2002] for details). Such very specialized algorithms do not fit into this general discussion of second generation distributed component systems.

2.8 Relation Transparency

There is little need to mention that links between data are an essential part of every modern document-, information- and knowledge-management system. However, there is need for discussion what the requirements for a modern implementation of the node-link paradigm are. From the discussion of the "holistic" view of the system in section 2.3 it is already known that essential types of applications built with component frameworks will surely be document-, information- and knowledge-management systems.

Therefore let us have a closer look at the requirements that such systems have, to derive the technical requirements for distributed component systems:

- It is clear that hyperlinks embedded in e.g. HTML-documents are not the solution we all are looking for (see [Andrews et al 1995]). Hyperlinks definitively have to be separated from documents (at least internally).
- It is also clear that hyperlinks have to be robust against moving the destination to a different location. In this case the hyperlinks have to point to the new location.
- It has to be possible to interlink arbitrary kinds of documents, no matter where they reside and no matter which type they have. And this is exactly the point, why the term hyperlink seems unappropriate. There is much more behind this requirement than one would suspect. What is definitively needed, is a general mechanism to define arbitrary kinds of relations between arbitrary components. By *arbitrary kinds* of relations things like a *link to* a destination or an *inclusion* or just an *interconnection* are meant. The list what a relation can represent is endless and depends on the needs of a concrete application. Relations cannot only represent navigational structure, they can also be used for internal structuring purposes, e.g. for combination of several components into one virtual component.

It should be clear that there is a myriad of examples how data can be connected. Considering the relation-topic from a more technical point-of-view, one very important group of features comes into mind, resulting in very essential requirements for distributed component systems:

Components can, amongst other things, also represent functional modules (as is the case with e.g. Java Beans). Such functional modules are combined in one way or another to make up whole applications. In our case the single components can be arbitrarily distributed across several computers resulting in a wholly distributed application. All transparency requirements that have already been discussed above also fully apply for this case. For example, if a functional module is moved from one location to the other this must not break the distributed application!

There is one more requirement that can be derived from the discussions in section 2.6 and section 2.7: there it was stated that addressing and navigation are technically and semantically two completely different mechamisms that have to be strictly separated. The solution for the dynamic location transparency problem is the use of globally unique handles. Relations are now the method of choice to implement navigation.

In fact, from the users' point of view, navigation always comes down to either moving around in a hierarchy (e.g. the subdirectory-structure of a filesystem) or in a graph (e.g. hyperlinks on the Web or symbolic links in Unix filesystems). In case of a hierarchy we have to deal with parent-child relations, in case of a graph we have to deal with directed relations from one arbitrary point to a different arbitrary point. Relations always represent some kind of *logical* structure. Addresses always represent a *technical* structure.

With these points in mind the requirements for a relation mechanism in second generation distributed component systems can be formulated as follows:

- For the sake of generality n:m-relations have to be used. In most cases only 1:1 or 1:n-relations will be needed in applications. Nevertheless there are situations where the general n:m-case applies (e.g. when interconnecting two version-controlled components). One can simulate n:m-relations by using many 1:1 relations, but this would cause avoidable overhead. Therefore, from now on the term *relation* in this paper is always understood to be an n:m-relation.
- The endpoints of relations are always attached to components. If endpoints of relations shall point *inside* components, e.g. refer to a paragraph in a document, this can be achieved as well. Two cases exist here:
  1. The endpoint of a relation may be a component that represents a part of a document, e.g. a paragraph.
  2. If the granularity of sub-components is not small enough, a relation has to point to something that is just part of a component. In this case additional information can be attached to the endpoint of this relation at issue that reflects this fact.

  Jumping a little ahead, the advantage of attaching endpoints to components is that robustness concerning component-movement problems can be achieved easily. If e.g. a paragraph of a document is represented as a component and the paragraph is moved inside the document, the relation automatically points to the new location of the paragraph in the document.

– Relations can be of arbitrary kind (directed, bidirectional, inclusion, etc.) and of arbitrary user- or application-defined type (inline-image, belonging-together, interesting-additional-information, etc.).
– Arbitrarily many relation-endpoints can be attached to a component
– Relations have to be robust against component-movement problems.
– It has to be possible that relations between relations exist.
– Internally relations have to be implemented bidirectionally, so that it is always possible to find all n + m endpoints of a relation. This requirement is an absolute (internal) necessity to fulfill the movement-robustness requirement.
– Relations and single endpoints of relations can have arbitrary meta-information attached.

Considering these requirements it becomes clear, why the heading *relation transparency* was chosen for this section: with the definitions that "everything is represented by components" and "relations interconnect arbitrary components" it is possible to define relations between arbitrary data, no matter if the data-format natively supports relations or not. If the underlying data-format supports relations they are passed on to the component transparently. If not, the relations are managed by the system and stored in a separate database. Arbitrary mixtures between implicit and explicit management of relations for one component are possible.

2.9 Replication Transparency

There are two main factors that make replication of components desirable:

1. If many users want to use one and the same component it can happen that either the machine where the component resides or even the network in this area become overloaded.
2. Network connections to a certain location may be slow from parts of the network.

Thus, since response time may be rather unsatisfactory, some sort of replication mechanism would be desirable. By using globally unique handles this can easily be implemented: resolving a handle can return an appropriate remote reference to a replica of the desired component rather than to the original. Therefore replication of components is fully transparent in a sense that requestors do not notice at all whether they obtain a reference to a replica or to the original.

Sofar this mechanism corresponds to a standard caching mechanism as can be found in every Proxy. The difference between simply caching a component or having a real replica is that caching is unsynchronized from the point of view of the original, while replication is not. Replication has to be implemented in a way that the original knows of existing replicas and can set them *dirty* if something changes in the state of the original.

From this point of view there exist three kinds of replicas, depending on the nature of the component itself and depending on the usage of the component:

Unsynchronized replicas: these are replicas, where synchronization is not necessary at all because the original component is stateless. The mechanism in this case corresponds to a standard caching mechanism without *dirty*-flagging. However one thing has to be kept in mind that forces real replication (i.e. the original knows about existing replicas): if a component is deleted, the replicas have to be deleted too. Therefore just for the case of component deletion either *close synchronization* or *loose synchronization* as described below are necessary. For this reason unsynchronized replicas may only exist in systems that do not allow object deletion. Hence such systems make only limited sense, but for the sake of completeness of the discussion this case is mentioned here.

Closely synchronized replicas: these are replicas, where updates of the internal state are essential for working with them. The problem with this kind of replicas is that delays in setting them *dirty* influences the result in an unacceptable way. Therefore it has to be made sure that the actual state of the original component is always reflected in the replicas. Especially when dealing with collaboration aspects like concurrent editing, close synchronization is the method of choice. It might be suspected that this means that the original has to be contacted anyway for each request and that therefore replication does not make any sense at all in this case. However, this is not really true. Timestamped requests together with replicated version-update information reduce network traffic for closely synchronized replicas enormously.

Loosely synchronized replicas: these are replicas, where delays in updating the internal state of components are not critical, as long as the delays can be kept within certain boundaries. Usually some seconds of delay, sometimes even minutes or hours could be considered uncritical. Just think of a standard WWW-server: when pages are changed it usually does not matter at all, if some users see the old page rather than the new one, even if the new page would already be available for some seconds. This kind of delay is quite usual and commonly accepted today if you consider all the caching mechanisms in proxies and in common browsers. However, one point has to be kept in mind when discussing loosely synchronized replicas: it has to be possible to force a lookup, if an update occured. With this additional forced synchronization that can be triggered by the replica, one can at least make sure to obtain an updated version if this is absolutely desired.

Speaking of replications and updates the first thing that usually comes into mind is the problem that a requestor obtains an outdated version of an object. However, also the opposite can be a problem: a requestor could obtain a version that is “too new”. Just think of the case that the network connection between requestor and replicating system is slower than the connection between the replicating system and the system that holds the original. Under certain circumstances it could happen that at the time when the request was sent, an older version was valid than at the time when the request arrived at the replicating system. If the replicating system then sends the newer version of the object rather than the one that was valid at the time when the request was sent, this could be a problem. For most applications it is ok or even desirable to always get the newest available version, for others, e.g. for collaboration purposes, it is not.

Therefore replication in a second generation distributed component system has to be implemented in a way that the behaviour can be adapted to the needs of the application. Different strategies have to be available to choose from, depending on the requirements.

3 Dinopolis - the First Second-Generation System

The need for systems covering the aspects discussed above led a team of researchers at the IICM to start an open-source framework called *Dinopolis* (see [Freismuth et al 1997], [Dallermassl et al 2000a] and [Dallermassl et al 2000b]). From 1997 on design and prototype-implementation phases have been going on until in 1999 version 2.0 of a system called *DINO* (**D**istributed **I**nteractive **N**etwork **O**bjects) was the core for *MTP* (**M**edical **T**elematics **P**latform, see [Aly et al 1998]). MTP is the first system implementing arbitrarily distributed virtual medical patient records. The first prototype of MTP was introduced at CeBit 1999 and due to the strong interest among medical institutions and doctors, phase 2 of MTP, the design and implementation of a production release of the system, started end of 1999. Since then a group of researchers and developers at German Aerospace and at the IICM have been working closely together on the design of *Dinopolis* as the first second-generation distributed component system, which will be the core for the production release of MTP.

The cornerstones of Dinopolis that make it a full-featured second-generation distributed component framework can be subsummarized as:

- Dinopolis is designed as a platform independent middleware system, fully written in Java.
- Due to its concept as a middleware system, Dinopolis is able to embed and combine arbitrary existing systems, such as databases, Web-Servers or also ORBs.
- Dinopolis implements a highly sophisticated component-model that fulfills all transparency aspects discussed in section 2.1, section 2.2, section 2.3 and section 2.4. Components can reside anywhere on the network or in arbitrary

embedded systems. Due to its design as a middleware system Dinopolis takes over component integration and management.
- Dinopolis implements a highly sophisticated addressing mechanism via globally unique component-handles that fulfills all the requirements discussed in section 2.5, section 2.6 and section 2.7. Handles are robust against component-movement which can e.g. happen due to restructuring of the distributed component space.
- Dinopolis implements a highly sophisticated relation mechanism that fulfills all the requirements discussed in section 2.8.
- Replication transparency as discussed in section 2.9 is made possible by Dinopolis' addressing mechanism.

Because a detailed description of the whole Dinopolis system would be far beyond the scope of this paper, this paper emphasizes the three most important aspects: the *component definition*, *globally unique handles* and *relation management*.

4 Definition: Component

Because the terms *component* or *object* have been used as buzz-words for a very long time, there exist many different and even contradictory definitions. In the following the definition of *component* that forms the basis of the Dinopolis middleware framework will be discussed.

In principle a component in a distributed component system is an addressable entity with the following properties:
- A component is addressable in a unique way via globally unique handles. This means that one handle is always resolved to exactly one and the same component, no matter when and how often it is resolved. It cannot happen that a component is replaced by a different one by accident, like it can happen in today's systems, if one component is deleted and a different component happens to get the same address at a later stage. If a component is deleted it is guaranteed that the handle will never be re-used again for a different component. A more in-depth discussion about handles can be found in section 5.
- A component can itself be a compound made up of several part-components. In this case also the parts fully correspond to the whole component-definition given here. In an OO-sense different models of composing components to a compound apply, e.g. derivation, inclusion, etc. With this feature arbitrary component-hierarchies can be modelled.
- A component encapsulates content. Content in this context is everything that can be considered data in a broad sense, e.g. a document, stream-data or whatever else could be state-information.

- Arbitrary meta-data (i.e. attributes) can be attached to a component. Meta-data can e.g. be of descriptive nature like *author*, *type* or *creation date*. Meta-data can also be dependent on certain applications that need to deal with the components, e.g. *display hints*, etc. For this reason meta-data is defined to be a tree-structured container of keys with values of arbitrary type that can be accessed through the keys.
- Arbitrary relations can be attached to a component or to parts of it. Relations can also e.g. be attached to meta-data. As an example there can be a relation from the *author* attribute to an address-record in a database that represents the author.
- A component can provide arbitrary operations. From an OO-point of view the operations can be seen as the *methods* of a component.
- Sometimes operations are not enough to deal with components, because it can happen that too much knowledge about the internals of the component could be necessary. For example a component could have its origin in a database that supports very special user access-rights. If an application would want to provide e.g. a graphical interface that would allow users to change access-rights, then the application would have to have the knowledge about the internals of the database. E.g. the syntax of the attributes to call a method for setting the rights correctly has to be known. For this reason components can also provide arbitrary so-called *services*. Services in the context of Dinopolis are GUI-objects that applications can request and that provide high-level user-interface functionality for special purposes that would otherwise require too much internal knowledge. Services deal with arbitrary user-interface libraries and their look-and-feel is configurable accordingly, but this is beyond the scope of this paper.
- Components have a standard, uniform interface representing access to their content, meta-data, relations, methods and services. Therefore applications need not know the internals of different components to deal with them. Part of the standard interface is also a possibility to ask components for their capabilities in a uniform way. For example one can ask a component whether it supports versioning.

A schematic view how application programmers see components according to the definition given here, is sketched in figure 1.

5 Globally Unique Handles

From the discussion of the requirements at the beginning of this paper we know that components have to be accessible through globally unique handles. These handles have to be robust against component movement and one handle always refers to one and the same component. Handles can also be stored, e.g. somewhere on a user's harddisk when bookmarking a component.

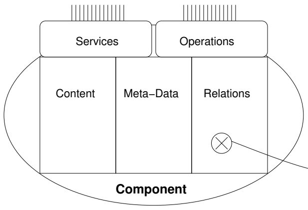

Figure 1: Schematic view of components

Considering these requirements it becomes clear that there are two ways to ensure consistency of handles: either moved components leave traces in the form of forwarders or a lookup service is implemented. The algorithm with forwarders does not scale at all considering a huge number of objects and a highly dynamic case. In addition, the requirement for component replication (see section 2.9) would not work with forwarding anyhow and would definitely require a lookup-mechanism.

Therefore a lookup-service has to be the implementation of choice. However, considering a huge number of handles in large and highly dynamic distributed systems, a naive implementation (e.g. a central lookup service) will not be enough, because it would not scale either.

The first idea that comes to mind to get control over the situation is to define hierarchically structured handles and treat them like hostnames are treated in DNS (see also [Mockapetris, Dunlap 1988]). With this approach the lookup-service is well distributable. Nevertheless there still exists a huge problem: we required robustness against object-movement, even if handles are stored "somewhere". Therefore, if a component is moved from one "domain" to another, either its handle would change or one lookup-service would have to take over control of the handle space of a different domain. Both approaches are not realistic.

It becomes even worse if we consider the case of a heavily growing system. At the beginning one lookup-service is enough, but as the number of objects in the system and the number of users of the system grows the lookup-service has to be split across two or more machines. Also the opposite case is possible and we have to deal with it in the case of the MTP Project: if a doctor representing a data-storing institution retires and the system would go offline, the data has to be stored in a different system, causing a "merge" of two systems. Besides it can happen that not only a simple merge of two systems takes place, but that the contents of the system going offline could even have to be split across several systems.

Thus, everything can move, components, parts of components (in the case of compounds), servers that store components and even lookup servers. Nonetheless globally unique handles have to remain stable and have to be robust against all dynamic changes that can happen!

For this reason we developed an algorithm called *DOLSA* (**D**istributed **O**bject of **L**ookup **S**ervice **A**lgorithm) that deals with arbitrarily granular, arbitrarily distributed lookup-servers and keeps handles stable, no matter which dynamic changes in the whole component- and lookup-service world take place. The detailed description of this algorithm can be found in [Schmaranz 2002]. Here is just a summary of its very basic principles:

– Globally unique handles always consist of three parts, which can be partially empty, if nothing has moved:
  1. The *Birthplace-ID* of the component. This is the part of the handle that always allows it to find its location. Therefore in a way this is exactly the globally unique handle that we are looking for and that must never change. However, just having this ID does not scale for huge numbers of objects and in the highly dynamic case.
  2. The *Moved-Birthplace-ID* of the component. This ID represents the new ID, if the birthplace lookup-service is no longer available and a different lookup-service has taken over control. If this "new" lookup-service is also no longer available and its responsibility is therefore moved again to a different server, this ID is overwritten by the actual one. Please note that overwriting this ID happens for scalability reasons, but it is not essential for resolving a handle. The *Birthplace-ID* can always be resolved. The algorithm also deals with the case that one Birthplace lookup-service can be split across several systems.
  3. The *Actual-ID* of the component. This ID represents the ID in the lookup-service that is responsible after a component was moved across the network.

– Each of the three parts of the ID described above consists itself of two parts: a *Lookup-Service-ID* and an *Object-ID* within the lookup-service.

– Lookup-services are hierarchically organized, but this organization is not reflected in their *Lookup-Service-IDs* to remain robust against changes in the hierarchy. The principle here is the same that led to the separation of relations and handles. The hierarchy of lookup-services makes sense for scalability reasons: a request to resolve a handle is always sent to the "closest" lookup-service. Lookup-services can cache resolved handles very similar to DNS servers and can give *authoritative* and *non-authoritative* answers. If a handle cannot be resolved locally, the request is passed further up the hierarchy until it can be resolved and the result is cached.

– To make sure that the distributed lookup-services can always be found, the top of the hierarchy is formed by a set of so-called *Master-Lookup-Services*.

– All IDs, *Lookup-Service-IDs* as well as *Object-IDs* are of arbitrary length in chunks of 64 Bits. This prevents the case of running out of free IDs, although this may seem unnecessary when using 64 Bits. However, there is the requirement that IDs must not be used twice and there are components that travel around a lot (like e.g. mobile agents). Thus they can effectively "eat up" lots of IDs and having no limit can therefore be essential.

6 Relations

As has been discussed, the most universal case of relations are n:m relations and for this reason they are implemented this way in Dinopolis.

Relations can interconnect arbitrary components or even other relations. There are enough examples, where at least one endpoint of a relation is a relation itself, e.g. a hyperlink that says "have a look at this link". As is the case with handles, also relations have to be robust against component movement. The simplest and besides the most logical way to achieve this, is to define the endpoints of relations by globally unique handles. The further logical consequence is that relations are components themselves. Relations being components in the sense of this paper result in a flexibility that cannot be found in any other system:

– Relations between components can be held anywhere and are not bound to the components' locations. Therefore it is possible to e.g. use relations for personal hyperlinks between documents that reside on the users' desktop-computers. Additionally those hyperlinks are kept consistent if documents are moved.
– Arbitrary type and meta-information can be attached to relations.
– Not only meta-information can be attached to relations, they can also provide methods and services for greater flexibility.
– Internally relations are multi-directional, the endpoints of a relation are sub-components and the relation-component is the enclosing composite. For consistency reasons, e.g. when restructuring the component-space it is necessary to find out which components are interconnected.

Because relations are components of their own that can be stored separately, it is possible to interconnect arbitrary objects that are not even aware of relations at all. For example it is possible to annotate video-streams or audio-streams. Even private annotations are possible that are not visible for others.

Typed relations with arbitrary meta-information also make it possible to have arbitrary many different navigation-paths through huge component-spaces without having to create many different sets of hyperlinked documents as would be the case in today's systems. As an example consider an e-learning application: re-using existing course-material and structuring it for different audiences by means of typed relations for navigation is an easy task to do. It is then even possible to switch back and forth between different navigational structures. This makes it easy to build adaptive courses, where navigation depends on the skills of the learners (see also [Dallermassl et al 2000c]).

One of today's buzz-words is *Knowledge-Management*. Without going into details of Knowledge-Management, one of the main goals of KM is to put information into context to make it knowledge. As knowledge is growing, one aspect of growth is the number of interconnections between different information chunks. The more interconnections between related topics exist, the better the knowledge-base. However, it does not always make sense to see all the interconnections. One and the same chunk of information can be interesting for different audiences, but from different points of view. As the number of interconnections grows and as the number of different points of view grows, it becomes necessary to have adaptive relations, so that users only find relevant knowledge rather than having to extract relevant parts themselves from a huge pile of interconnections. There are many different examples where a flexible relation-mechanism is essential.

7 Conclusion

According to the motto *"a good component system is one that the application programmers don't notice"*, Dinopolis is trying to implement all transparency-aspects discussed in this paper. Distributed component systems will eventually become some sort of high-level operating systems that serve as a platform for all different kinds of applications. If this is the case it would absolutely be desirable to standardize such frameworks. For this reason and also because different people have many different ideas about what such a platform should provide, Dinopolis is an open-source project and the results are available for everyone free of charge.

Dinopolis is not intended to be a huge monolith. Just the opposite is true: the core of the system is a very slim middleware layer providing the basic functionality of globally unique handles, relations and a highly sophisticated object-model. Everything else is grouped around this core in the form of modules that can be loaded dynamically during runtime. Therefore the system is adaptable for everybody according to the special needs of different applications.

One of the applications that require the implementation of a very robust system with highly sophisticated access and security-mechanisms is MTP that the IICM develops together with German Aerospace. The security-, reliability- and robustness-requirements for medical applications are extremely high because all the data in the system is extremely sensitive. Therefore Dinopolis is not developed "quick-and-dirty" but very structured with a very detailed design-phase and throrough documentation.

Because we want to build a platform that can be used for as wide a range of different applications as possible, all ideas for necessary or desired modules that can be grouped around the core of the system are very welcome. If you have ideas or questions please have a look at `http://www.dinopolis.org` or feel free to contact us via email at `contact@dinopolis.org`.

References

[Andrews et al 1995] Andrews K., Kappe F., Maurer H., Schmaranz K.: *On Second Generation Hypermedia Systems*, Proceedings ED-MEDIA 95, Graz (1995), 75–80.

[Aly et al 1998] Aly F., Bethke K., Bartels E., Novotny J., Padeken D., Schmaranz K., Schwartmann D., Wilke D., Wirtz M.: *Medical Intranets for Telemedicine Services: Concepts and Solutions*, Proceedings G7 Meeting "The Impact of Telemedicine on Health Care Management", Regensburg (1998), available online at `http://www.uni-regensburg.de/Fakultaeten/Medizin/Uch/g7/program/mon.htm`.

[Berners-Lee et al 1994] Berners-Lee T., Masinger L., McCahill M.: *RFC 1738: Uniform Resource Locators (URL)*, available online at `ftp://ftp.internic.net/rfc/rfc1738.txt`

[Dallermassl et al 2000a] Dallermassl C., Haub H., Maurer H., Schmaranz K., Zambelli P.: *Dinopolis - A Leading Edge Application Framework for the Internet and Intranets*, Proceedings WebNet 2000, San Antonio, TX (2000), 111–116.

[Dallermassl et al 2000b] Dallermassl C. Haub H., Krottmaier H., Schmaranz K., Zambelli P.: *Using Highly Sophisticated Middleware for Building Arbitrarily Distributed Teaching Environments*, Proceedings ICCE/ICCAI 2000: Learning Societies In The New Millennium: Care ativity, Caring & Commitments, Taipei (2000), 1439–1442.

[Dallermassl et al 2000c] Dallermassl C. Haub H., Krottmaier H., Schmaranz K., Zambelli P.: *Adaptive Learning Environments*, Proceedings ICCE/ICCAI 2000: Learning Societies In The New Millennium: Care ativity, Caring & Commitments, Taipei (2000), 1443–1446.

[EJB] *Enterprise Java Beans Technology*, electronically available at `http://java.sun.com/products/ejb`.

[Freismuth et al 1997] Freismuth D., Helic D., Meszaros G., Schmaranz K., Zwantschko B.: *DINO - Distributed Interactive Network Objects – The Java Approach*, Proceedings Ed-Media '97, Calgary (1997), available online at `http://www.iicm.edu/liberation/iicm_papers/edmed97/dino.html`.

[Mockapetris, Dunlap 1988] Mockapetris P., Dunlap K. J.: *Development of the domain name system*, Proceedings ACM SIGCOMM 1988, Stanford, CA (1988), 123–133.

[OMG] *The Object Management Group's Home page*, electronically available at `http://www.omg.org`.

[RMI] *Java Remote Method Invocation*, available online at `http://java.sun.com/j2se/1.4/docs/guide/rmi`.

[Schmaranz 2002] Schmaranz K.: *DOLSA - A Robust Algorithm for Massively Distributed, Dynamic Object-Lookup Services*, submitted to J.UCS.

[Terry 1984] Terry D. B.: *An analysis of naming conventions for distributed computer systems*, Proceedings ACM SIGCOMM 1984, Montreal (1984), 218–224.

[Voyager] *ObjectSpace's Home page*, available online at `http://www.objectspace.com`.

[Znati, Molka 1992] Znati T. B., Molka J.: *A Simulation Based Analysis of Naming Schemes for Distributed Systems*, Proceedings of the 25th annual Symposium on Simulation 1992, Orlando, FL (1992), 42–51.