

# Dinopolis - A Leading Edge Application Framework for the Internet and Intranets

Christof Dallermassl (cdaller@iicm.edu),  
Heimo Haub (hhaub@iicm.edu),  
Hermann Maurer (hmaurer@iicm.edu),  
Klaus Schmaranz (kschmar@iicm.edu),  
Philipp Zambelli (pzamb@iicm.edu)

March 20, 2000

## Abstract

Dinopolis is at the moment the most universal and extendable application framework available. It is fully written in Java and fully compliant to today's application standards like Corba, RMI, DOM and XML. As a middleware system it provides access to arbitrary existing systems and supports combination of external systems in a way that they benefit from each other, although they do not even know about each other. The Dinopolis approach described in this paper makes it very easy to integrate even very proprietary systems and to provide a portal to a distributed, (globally) shared resource space. Dinopolis is accessible with arbitrary clients (e.g. Netscape, MS-IE) due to the extendable gateway philosophy, not to speak about Corba and RMI accessibility for application programmers. In this paper the motivation for Dinopolis is explained as well as the cornerstones of the design and implementation are briefly discussed.

**Keywords:** Internet, Intranet, Distributed Object System, Middleware, Application Framework, Java

## 1 Motivation

Although corporate Intranets, application servers and portal systems have been buzz-words for quite some time now and although most organizations are having some kind of an Intranet structure, it is nevertheless worth to take a deeper look at today's solutions: what is called a corporate Intranet mostly consists of one or more HTTP servers and a mixture of CGI-scripts and servlets to make some information from other sources available on the Intranet. Access-logic for the different systems is partially hidden behind the scenes and therefore access is easier for the users. Sometimes so-called portal servers are providing a single point of access to several resources of the corporation. Nevertheless, there is

usually no combination of different available systems in a sense that they add value to each other.

To make this fact clearer let us consider an organization that is running an SQL server for administrative information like employees' data, general project data and bookkeeping data. In addition, suppose that project management is done using a standard system like e.g. MS-Project, software developers are using CVS for version-controlling source code, source code documentation is stored in the filesystem available to others via NFS, for bugtracking and software support a self-made full-custom proprietary software package is used; and even the shared calendar is a proprietary solution. If there is any portal system available in the company at all, then it is usually a so-called application server or enterprise server that provides HTTP access to some of the information space. The SQL server is accessible via servlets and the project plans are made accessible if they are inserted into the server by hand. Maybe also the bugtracking- and support-systems are accessible via HTTP and nobody has to care about the access logic for the different systems. Clearly, such a portal is a great advantage compared to having several different systems requiring different software to access them and requiring lots of knowledge about the individual systems. One of the most advanced systems is the Hyperwave Information Portal (short *HIP*, see also [Hyperwave]). However, even such sophisticated portals cannot be seen as the proper long term solution. Several further important aspects have to be taken into account:

- Although systems are accessible through one single portal they are still somehow treated as stand-alone systems. No real *combination* of the systems is achieved because they do not know about each other and therefore they cannot benefit from each other. Combination in this context means that e.g. an information system serving documents without meta-data can be combined with a database serving meta-data. The portal as a result serves documents with meta-data transparently by taking the documents from one system and the appropriate meta-data from the other.
- No stable address-space is provided by portals, therefore restructuring the underlying information space is a rather difficult affair.
- No globally unique address space is provided and usually URLs with proprietary parameters interpreted by special CGI scripts or servlets are used for access. Therefore addresses are only valid for one portal, rather than being globally usable. It would be desirable to use handles that are globally unique and can be resolved by different running portals. This would enormously ease scalability and load balancing.
- Portals themselves can usually not be distributed in a sense that several portals integrate different systems and are combined to provide a transparent distributed resource-space accessible as a unit.
- Usually only systems providing (passive!) information can be accessed through the portal. Active systems supporting mobile agents, communication and collaboration features cannot be fully utilized.

From our point of view an organization's knowledge and information space has to be considered as a huge arbitrarily combineable and easily accessible

distributed resource space. Systems embedded in this resource space have to gain value from each other, users should have different ways available to access the whole resource space and applications should easily be able to utilize this resource space for their own purposes. To get back to the example above: wouldn't it be nice to be able to have links between project-management data, employees' data of the programmers and the source code they have written? Additional links between the bugtracking system and the appropriate versions of the source-code combined with the support system would ease software support enormously. If this is then also interlinked with the shared calendar, progress tracking and timetables for future developments are much easier to maintain. Also the shared calendar can be automatically linked e.g. to the organization's email system so that all coordinators are automatically alerted of changes in schedule. Active trigger elements and software agents can ease coordination of huge and complex projects.

Just as a tiny example an agent (see also [Logan, Sloman 1999]) could be sent into the system to coordinate an urgent meeting between a sales manager, the head of software development and some programmers working on a particular project. This agent then has a look at the calendar of the single persons as well as their scheduled time for working on different tasks and their deadlines and tries to find out the next possible date for a meeting. This agent also takes into account the priorities of the work to be done and the priority of the meeting and then automatically sends email to all participants. The participants then can decide to either accept the proposed date or ask the agent to look for a different one in which case the agent looks for the next possibility.

Today there are several highly sophisticated enterprise- and application-servers existing like e.g. Netscape's application server, Oracle 8i, IBM's WebSphere and NetDynamics by Sun just to mention the most important ones. Also a few distributed object systems like ObjectSpace's Voyager provide some of the functionality required for a distributed resource space. Several middleware libraries for system integration help the developers to gain easy access to external resources.

All of the systems mentioned above have their advantages, but none of them really provides the functionality needed to fulfill the vision of a distributed resource space that is outlined above. What is really needed is a middleware layer that is able to embed existing systems, allows to combine their features and provides a virtual data- and application model to access them in a uniform way. Please keep in mind that uniform access in this case does not only mean uniform for users like e.g. HTTP portals allow. Uniformity shall also be provided for applications in a sense that they do not have to be aware of the underlying systems at all. Additionally the whole middleware layer has to be open to allow integration of arbitrary systems rather than to support just a few standard ones. Last, but not least, the layer has to be platform independent and open to existing standards like Corba and Java RMI to easily allow standard applications to access it and gain all the benefits.

With these requirements in mind a team of 8 researchers started to design and develop a Java middleware system in 1997 called *Dino* (see also [Freismuth et al. 1997]). After version 3.0 of *Dino* was successfully used for a medical telematics platform called *MTZ* (see also [Freismuth et al. 1998]), which was presented at CeBit 1999, the team decided to use the experiences gathered with *Dino* throughout the years, to make a complete redesign, and to

make the resulting system named *Dinopolis* publicly available under an open source license (see also [Dinopolis Developer Community Home Page]). In the following sections the cornerstones of Dinopolis will be described – cornerstones that make it a unique framework providing functionality not available with any other system up to now!

## 2 The Dinopolis Distributed Application Framework

From the very beginning four main design principles formed the credo of Dinopolis development:

- Dinopolis is a very modular, extendable and platform-independent system rather than a huge proprietary monolith. The base system providing the information model is very slim.
- Dinopolis allows to integrate and combine arbitrary systems, no matter if they are only dealing with passive content or also with active content as “real” distributed object systems do.
- Dinopolis is an application framework that supports major industry standards. It provides access to the underlying distributed information space in a fully transparent manner for users as well as for applications utilizing it.
- Dinopolis systems can be arbitrarily distributed across the network. Nevertheless all handles used to access objects within the framework are globally unique and stable. Therefore once a handle is obtained, the object that it refers to can be moved to arbitrary locations, even across the network but the handle will still be valid.

To achieve platform independence to the maximum extent possible today Dinopolis is written in pure Java. Since Java allows classloading on demand this decision also provided the key to make the base system that implements the Dinopolis virtual address- and relation-space very slim. All additional functionality is grouped around this kernel in a modular way without making it a huge monolith by use of factories (see also [Gamma et al. 1995]).

The concept of the virtual address- and relation-space is general enough to be able to integrate basically all systems that allow some kind of addressing of objects. No special requirements for the kind of addressing have to be met for integration. For this reason Dinopolis addressability is supported for all systems known today from simple filesystems to databases or other information systems as well as for distributed object systems. Integration of systems is done by means of so-called *embedders* that provide addressing as well as arbitrary additional functionality of the underlying systems. With this concept the superset of functionality of all systems is available for the applications rather than the subset as in other comparable middleware approaches.

Since Dinopolis is a distributed application framework it has to be useable in different ways. First of all it can simply be taken as a Java library that applications can utilize. On the other hand one of the Dinopolis modules allows

it to run a server that supports RMI and Corba for already existing applications that want to benefit from its features without rewriting the application. Last, but not least, the standard distribution also contains external access gateways (=EXAGs) to allow access to the information space via HTTP, FTP and Telnet. Other EXAGs like SMB, NFS, etc. will follow in the near future.

Embedders as well as EXAGs are easy to write, therefore third parties can tailor the system according to their needs. Because Dinopolis is open source it can be expected that many different embedders and EXAGs for all commonly used systems will be publicly available soon. An example for a custom embedder would be to integrate a proprietary bookkeeping system that has been around in an organization for ages, thus making it part of the globally shared resource space. Depending on the complexity of the underlying system, embedders are usually very easy to write. Providing addressing for objects in the underlying system is enough to integrate the external system and make it available for Dinopolis. As soon as a system is embedded all the benefits of stable object handles, arbitrary relations and links, special attributes, etc. apply for this system implicitly.

As an example for a custom gateway take somebody wanting to use Dinopolis with a Gopher client. In this case an EXAG for Gopher access can be written within a day (and will very likely be available in the standard distribution soon), because implementation of the slim Gopher protocol is the only work to do. The rest is a simple 1:1 mapping.

Due to the fact that Dinopolis provides access to objects by means of stable handles rather than instable URLs (see also [Andrews et al. 1995]) huge distributed resource spaces can easily be administrated. It is no problem to move objects around in the resource space, the handles are globally unique and stay valid. Beyond moving objects without losing stability, it is even no problem at all to change the underlying embedded system and still have valid handles! This fact is even more important as the following example shows: Take a small organization that maintains some files for their employees' data. If this organization grows it is very likely that this way of storing data has to be replaced by a powerful database for administrative reasons. Using Dinopolis as a front end to access the data completely decouples the application from the underlying system. This allows to transparently embed the database instead of the filesystem and the handles still remain stable. Applications accessing the Dinopolis resource space do not even notice the change!

However this is still not the whole story - Dinopolis handles are globally unique in a way that, no matter from which running Dinopolis system resolving of a handle is requested, it will result in the same object. This can, but need not mean that it refers to the same physical location, highly sophisticated mechanisms of the *Distributed Dinopolis Communication Protocol* (=DDCP) allow transparent replication and mirroring of resources. DDCP itself is a high-level protocol which utilizes RMI or Corba (or if desired also other protocols) as a transport layer.

Handles also need not refer to one single object in an embedded system. Due to the sophisticated embedding and combination mechanism of Dinopolis, handles can refer to a combination of objects. As an example a simple filesystem can be embedded to store arbitrary chunks of information. Besides an SQL server is embedded to store meta-data for the information chunks and make it searchable. Additionally a directory service like LDAP is embedded to store

logical combination-relationships of documents. A handle then can point to one *logical* document that consists of several information chunks and several sets of meta-information as defined in the directory service. Requesting the object for such a handle then transparently returns one single combined content object representing the logical document. Even if chunks of such a logical document are moved, the handle for it is kept stable.

Last, but not least, the Dinopolis framework supports a highly sophisticated and highly modular security architecture. This architecture supports the use of security models implemented by embedded systems as well as to wrap an arbitrary security model around embedded systems. This allows it to incorporate authorization and authentication mechanisms even for systems that do not support such mechanisms at all. As is the case with data also the security policies of several systems can be combined to one logical policy. However a detailed discussion about the complex security model would be beyond the scope of this paper, so we will not discuss it in detail here.

### 3 The Dinopolis Addressing, Relation and Content Model

Much has been said about stable addressing and arbitrary relations between objects up to now. Now it is time to discuss the model that is implemented in Dinopolis to make this possible. The most important design point in the Dinopolis system is to completely decouple addressing from relations and content. Addresses in Dinopolis are wrapped by handles and are used to gain access to objects. It is not possible to *navigate* through the address space. That's what relations are good for: relations can be of arbitrary type, they can exist between arbitrary objects and they allow navigation in the resource space. All objects that can be accessed are considered *content* from the computer scientists' point of view and content is always presented in form of DOM trees. Therefore Dinopolis can also be seen as what is widely called an XML system today.

This might not sound great or new, therefore let us have a look at existing systems to see the difference:

As our first example let us take a simple filesystem. In this case the path to a certain file or subdirectory is the address of this object. However it is not only the address! It also implicitly contains parent-children relations that can be used for navigation. Therefore changing the address of a file by moving it around also changes relations. Although this is the semantic behaviour of a filesystem that everybody got used to, it also presents several problems because it is not possible to differentiate between *logical* and *physical* parent-children relationships. Usually subdirectories are used for logical structuring reasons, but if the disk becomes full and data has to be moved to a different location the physical necessity breaks the logical structure.

On a standard WWW server hyperlinks are used to model relationships and therefore the problem mentioned above is hidden behind the scenes. So far, so good, but this just moved the problem a little, because here hyperlinks are physically embedded in documents. Therefore relations and content are mixed up in this case, which makes the situation even more difficult: moving a document physically on a WWW server means to edit all documents with

hyperlinks pointing to the document that is moved.

In Dinopolis handles are used to access objects. These handles encapsulate the physical addresses following the URN principle. Due to internal algorithms the handles remain stable, even if the objects are moved. Relations between objects can be of arbitrary type (e.g. parent-children, hyperlink, etc.) and are kept separated from addresses and content in that they refer to stable handles. Therefore relations are also not affected by restructuring or even by a completely different address space due to underlying system changes. Content is considered a logical rather than a physical entity and is always presented in the form of a DOM tree that can be made up of arbitrary many different information chunks. Information chunks are then subtrees of the content object's DOM tree. This strict separation makes it possible to have a portal that really *combines* all the data and features of underlying embedded systems rather than just having a single point of access to separated systems.

## 4 Dinopolis Embedders and External Access Gateways

Now that we discussed the internal Dinopolis addressing, relation and content model, we can move our attention to the last cornerstones of Dinopolis: Embedders and External Access Gateways (short *EXAGs*). Embedders form the lowest layer of the Dinopolis system and are used to provide access to existing systems. *EXAGs* form the highest layer of Dinopolis and are used to provide access to the system for arbitrary clients.

The task of an embedder is to map the addressing scheme and the relation scheme of an external system to Dinopolis' internal representation as well as to provide objects of the external system as DOM trees. Because of the strict separation of the single elements in Dinopolis this is an easy task as the following example of a filesystem embedder will show:

Files in a filesystem are addressed by paths. Therefore a path to a file is handed over to the Dinopolis system as a so-called Dinopolis Object Handle (short *DOH*). Please note that this handle is different from the stable handles we discussed above, but a detailed explanation of the mechanism would be out of scope of this paper. Implicit parent-children relations in the filesystem used for navigation are handed over to Dinopolis in form of Relation Objects (short *ROs*). These relations are different from the internal stable ones, but they are used for stable access. However this mechanism is again out of scope of this paper and therefore it will not be discussed more in detail. Finally files are wrapped into Dinopolis Content Objects. A content object is always a DOM tree, so it depends on the type of file, how the tree looks like: it can be a single node in case of e.g. an image, but it can also be a "real" tree in case of e.g. an XML file. In Dinopolis factories are provided for content handlers that can deal with certain types of data, therefore the embedder does not have to care about the DOM model itself.

It can easily be seen that these simple requirements concerning addresses, relations and content can be fulfilled for more or less all different kinds of systems very easily. Therefore embedders for different systems are light-weight and can be implemented quickly, even by third parties.

The task of an EXAG is to implement the server part of the desired access protocol (e.g. HTTP, FTP, etc.) and provide the client with the desired data. Usually the only task to perform for an EXAG, as far as Dinopolis is concerned, is to map the requests to handles and return the requested objects to the client in a reasonable format. The Complexity of an EXAG is therefore defined by the requirements of the external access protocol and will in most cases more or less be a 1:1 mapping to Dinopolis' addressing, relation and content model.

## 5 Conclusion

Although a lot of the design decisions and algorithms could not be explained in depth in this paper, we hope that it became clear that Dinopolis is an application framework with functionality not available anywhere else. It is easily extendable by embedding new systems and combining them with others. Access is always performed by stable, globally unique handles, which makes administration and writing applications much less cumbersome compared to today's systems. Dinopolis is easily accessible with arbitrary clients by providing different EXAGs. Finally it is fully compliant to today's distributed object standards like Corba or RMI and using DOM/XML as the internal content model makes it easy to deal with all different kinds of resources.

Therefore it does not only have functionality not available anywhere else, it also has accessibility and extendability that goes far beyond other available systems. Since it is also fully written in Java it is as platform-independent and universal as today's techniques allow.

As a last plus-point let us again mention that Dinopolis is open source and available free of charge for everybody wanting to benefit from its features.

## References

- [Andrews et al. 1995] Andrews K., Kappe F., Maurer H., Schmaranz K.: *On Second Generation Hypermedia Systems*; Proceedings Ed-Media '95 (1995).
- [Dinopolis Developer Community Home Page] Dinopolis Developer Community: *Dinopolis*; available at <http://www.dinopolis.org>.
- [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, (1997).
- [Freismuth et al. 1998] Freismuth D., Schmaranz K., Zwantschko B.: *Telematics Platform for Patient Oriented Services*; Proceedings WebNet '98 (1998).
- [Gamma et al. 1995] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns - Elements of Reusable Object-Oriented Software*; Addison Wesley (1995).
- [Hyperwave] Hyperwave: *Hyperwave Home Page*; available at <http://www.hyperwave.com>.

[Logan, Sloman 1999] Logan B., Sloman A.: *Building Cognitively Rich Agents*;  
Communications of the ACM, 42 (1999), pp 71–78.