Object Distribution Framework (ODIF)

A Framework for Building Distributed Object Systems

Markus Voelter, voelter@acm.org

The Object Distribution Framework (ODIF) is a lightweight framework which applications to be transparently distribute their objects over nodes in a network. ODIF is based on the relay of method invocations. It distinguishes between the interface which is distributed and the algorithm that determines how method invocations are relayed. These algorithms can include load balancing, queueing, redundancy etc. ODIF uses CORBA and is currently implemented in the Java programming language.

0 Introduction

Today's applications are usually developed as distributed object systems or componentware. That means, that the modules of a software system are decomposed into objects which are then distributed over the nodes in a network. This distribution should be as transparent as possible for the programmer. Except from some additional error conditions and network latency, the programmer should not notice, that the objects his application uses are not located in his own object's adress space. Furthermore, the programmer should not need to care about the operating system underlying a remote object or the programming language in which it is implemented. Only the object's interface should be relevant.

To achieve this, programmers can use middleware technology such as CORBA. CORBA defines a software bus over which distributed objects can exchange messages (i.e. invoke methods) with each other. The client object need only know the server object's interface definition (which is given in IDL, CORBA's Interface Definition Language) and the object's identity. CORBA thereby solves the basic distributed object communication problem.

However, in many applications this is not enough. Because distributed object offer new chances and yield new errors, it is often necessary to introduce load balancing or redundancy features. This is where ODIF comes into play. ODIF provides a way to enhance CORBA systems with these features. The object implementations need not be changed at all. ODIF introduces Relay objects which are responsible for relaying method calls from the client object to the server object. These Relay objects can use arbitrary algorithms to relay the method calls. The algorithms are generic and can be used with any interface. ODIF provides GUI based applications to configure systems.

The remainder of the paper is organized as follows: Section 1 describes the problems ODIF solves, sections 2 and 3 describe the solution in detail and section 4 shows the tool support ODIF provides. Section 5 illustrated future improvements, 6 is a brief comparison of ODIF with other solutions available today, and section 7 is a summary of what has been achieved with ODIF.

1 The Problem

When building distributed applications, it is necessary to take care of the new kinds of errors which creep into the system, and to take advantage of the new possibilities distributed systems provide.

1.1 New kinds of errors

A distributed object system is made up of several discrete objects residing on separate nodes (computers) connected by a network. These objects can be divided in two categories:

- **Server objects:** Provides services to other objects in the network. The services are specified by the object's IDL interface.
- **Client objects:** Use the services servers provide by invoking the interface methods.

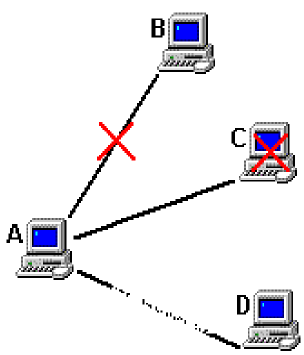

Illustration 1: Possible errors in distributed object systems

So, basically, distributed object systems are network systems. This means, that the typical errors which show up in network systems can happen here too:

- **Network failure:** If node A talks to node B and the network connection between node A and B goes down, node A can no longer talk to B and vice versa.
- **Node failure:** If node A talks to node C and node C fails, the communication can no longer be continued.
- **Network latency:** A network can become slower over time, either because of too much load or because of a physical failure in the cable. In either situation, it may make sense to consider the network connection as broken and to find an alternative.

1.2 Advantages of Distributed Object Systems

Of course, a distributed object system has many advantages, not just new possible errors. After all, that the reason for their popularity.

- **Load balancing:** It is possible to have multiple copies of the same Server object in the network. The client decides on which of these objects it invokes the service methods. This decision can happen either at setup time when the client connects to a server object, or before each method invocation. Especially in the latter case the client's decision should be based on some heuristics[1] to determine which server object to use. This can either be a server-independent heuristic (e.g. sequential use one of the objects, B, C, D, B, C, D, B, C, D, ...) or the client can query the server about its current system load and select the server with the lowest.
- **Redundancy:** If a server object becomes too slow or is not reachable anymore (because of the errors descibed in section 1.1), the client can use other copies of the same object residing on other nodes.
- **Queueing:** Method invocations can be queued and handled asynchronously.

1.3 Programming and configuration

The features described above are certainly useful. But there is an additional requirement: the features should not impose an additional burden on the programmer – they should be available transparently. In detail, this means:

- The programmer of the server object should not need to take care of these features. He just implements the server methods with the same business logic that would required for ordinary CORBA server objects.
- The client application programmer should not notice, that (or which) of these features are provided by the system and how. The client application programmer just gets himself an

1 heuristics: „rule of thumb“ that helps a client with a decision. Here, it determines which server object to use.

object reference (from a nameservice, etc.) and uses it.

- There should be no additional programming overhead to implement the features described in 1.2. The available algorithms should work independently of the the interface which is distributed.

Of course, there must be someone who configures the system in the way it is desired. This is the job of the system configurator (or administrator). He does not take care of how the interface's methods are implemented in the servers. He just defines the relationships between the objects available on the network and the algorithms used to implement the features of section 1.2. Ideally this should be possible without any coding. A graphical configuration tool would be useful.

1.4 Additional requirements

Of course, all the redundancy and load balancing features are useless if there is a central weak point during runtime that orchestrates the system and therefore must not fail. It is acceptable to have a tool/server/program which helps to initialize the distributed system, but once the system is up and running, this central tool must not be necessary anymore.

1.5 ODIF's current state

The current ODIF implementation is intended to be a proof-of-concept prototype. Some error checking and performance optimazations have not been implemented. That's also the reason why there are no performance considerations in this paper.

2 ODIFs approach

This section sketches the way in which ODIF solves the problems described in section 1.

2.1 The Relay metaphor

The central idea in ODIFs technology is the following:

All the features descibed in section 1 can be implemented by forwarding method calls according to some algorithm.

ODIF therefore defines an entitity called a *Relay object* which is responsible for relaying method calls to *Server objects*. A Server object's Relay object implements the same interface as the Server object, but it implements the methods by relaying (forwarding) the invocations to the Servers. The strategy used to determine which invocation should be forwarded to which server, and when, is determined by the so-called *Relay algorithm*. Illustration 2 shows this.

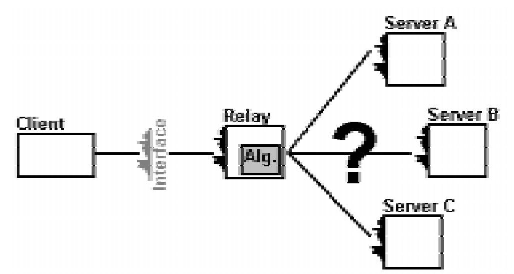

Illustration 2: The principle of relay objects

The following examples show how the Relay metaphor handles the features described in section 1:

- **Sequential load balancing:** Each time a Relay's method is invoked, the Relay chooses a different Server to which it forwards the call. It iterates the list of available Servers sequentially. There is no server dependent heuristics. However, this can be an effective load balancing strategy, especially if the invocations all take roughly the same time and the total capacity of all Servers is known to be enough.

- **Queueing:** The Relay object buffers the method calls and fowards them as soon as server objects become available. Internally it maintains a queue of incoming invocations.

- **Redundancy:** The Relay uses some server object A as long as this object is available. If an error occurs while communicating with this object, the object is assumed to be dead and a new object is used for future invocations, given that one is available and can be found via a nameservice, etc.

ODIF allows chaining of Relay objects. For example, a Relay which queues method invocations and implements load balancing can be implemented either as one Relay object with a special algorithm which performs both tasks, or it can be implemented by chaining together a Relay object with a queueing algorithm and one with a load balancing algorithm. These two Relays can reside in the same process or they can be located on different nodes.

2.2 Timimg assumption

There is one basic assumption underlying ODIF. The execution of a method should take significantly longer than its invocation process, even if multiple chained Relays are involved in the invocation. This is because ODIF imposes some additional overhead on method invocations. Applications in which ODIF can be used effectively are complex calculations, database queries, or report generation. Use of ODIF is not very useful for applications where only a short time is necessary to execute the methods on the Servers. ODIF is also not intended for real time applications.

There is a second reason why this timing assumption is important: Every call to a server object has to go through one central Relay object. If it takes a long time for the Relay to forward the invocation, or, if there are many invocations because the Server's method execution time is very short, then these Relays become the bottleneck we want to avoid. There is no use in load balancing the Servers when the Relay is so slow that none of the Servers ever gets any significant load.

3 ODIF in detail

With the help of an example, this section shows the components of an ODIF system. A **System** in ODIF terminology is a set of Server object together with the Relays. Some more definitions follow:

The **Interface** is the Server objects IDL interface. It defines a set of methods which the client can invoke. **Server** objects implement these interface with „working“ code. The **Client** uses the services (i.e. invokes methods) of the Servers. The client only sees the interface and does not know who implements it or how it is implemented. So far, this is the very same process as with standard CORBA. ODIF introduces two new components:

The **Relay** object is also a server object in that it implements the interface. But it implements the interface in a way specific to Relay objects: All incoming method calls are forwarded to one or more Server object according to some preconfigured algorithm.

The **Algorithm** is a piece of code that determines how an incoming method invocation is handled. Algorithms are always used within Relay objects.

3.1 The StringPrinter example

Imagine a very simple interface, which provides one method to print a String:

```
  // IDL
interface StringPrinter : ODIFBase {
  void print(in string s);
};
```

The IDL interface inherits from ODIFBase to provide some basic utility methods (the details of which are not considered here). It has one method, print(), that takes the string that should be printed as a parameter and returns nothing.

From this interface, the IDL compiler creates the usual CORBA stubs and skeletons. The server implements the skeleton in the usual way. This results in Java class that looks somehow like the following:

```
   // Java
package de.voelter.odif.test;

import java.util.*;
import java.util.*;
import de.voelter.odif.ODIFBase;

public class StringPrinterImpl
  extends _StringPrinterImplBase
  implements StringPrinter, ODIFBase {

     // method that prints a String to stdout
   public void print(String s) {
         // print string to stdout
       System.out.println( "StringPrinterImpl: "+s );
         // sleep 1 second, so that
         // method takes longer
         // (timing assumption)
       try {
          Thread.sleep( 1000 );
       } catch ( Exception ex ) {}
   }
}
```

Until here this has been standard CORBA/Java work.

The next step is to create the Relay objects. These are instances of a specific Relay class, a class that also implements the StringPrinter interface and which is generated automatically from the Server.

```
   // Java
package de.voelter.odif.test;

// import statements

public class StringPrinterImplRelay
  extends de.voelter.odif.test._StringPrinterImplBase
  implements de.voelter.odif.test.StringPrinter,
           de.voelter.odif.ODIFBase, RelayClass {

     // reference to the relay algorithm
   RelayAlgorithm algorithm = null;
     // list of destinations
   Vector destinations = new Vector();

     // implementation of print
   synchronized public void print(String p0) {
       CallData callData = new CallData( "print" );
       callData.addParam( p0 );
       algorithm.handleCall( callData );
   }

     // call method on server
   public void invokeMethod( ODIFBase dest,
            CallData cd ) throws Exception  {
```

```
      StringPrinter no =
        StringPrinterHelper.narrow( dest );
      Method destMethod =
        no.getClass().getMethod(
          cd.getName(),
          cd.getParamTypeArray()
        );
      destMethod.invoke( no,
        cd.getArgumentsArray() );
    }

}
```

A short description follows:

- The class implements the StringPrinter interface so that it „looks like“ an ordinary StringPrinter server.
- It has a reference to a RelayAlgorithm. This algorithm is responsible for determining which of the desitinations‘ methods gets invoked. (The algorithm reference is set up during Relay startup, the necessary code is ommitted here).
- Every Relay has a list of destinations to which the methods invocations are eventually forwarded. (The destinations reference are setup at Relay startup, the necessary code is ommitted here).
- The next important thing to note is the print() method, the implementation of StringPrinter’s interface. The method stores the information on itself, together with its arguments, in a CallData object and then passes this CallData object to the algorithm, which determines what to do with it. The important observation here is, that the algorithm only handles CallData objects, independently of what calls are stored in the object. Basically this is the way ODIF separates what is distributed from how this is done.
- The algorithm has to invoke methods on the destination Server objects. This process cannot be implemented completely by reflection and dynamic invocations in Java, so there is another method, invokeMethod(), in the Relay class. It does three things:
    1. Downcast the destination object to the interface required, here StringPrinter.
    2. Retrieve the method object from the destination object’s class using the invoked method’s name and parameter array.
    3. Invoke this method with the destination object and the arguments array.

Please note that this Relay class does not need to be programmed, it is automatically generated from the the Server object. This process is described in section 4.

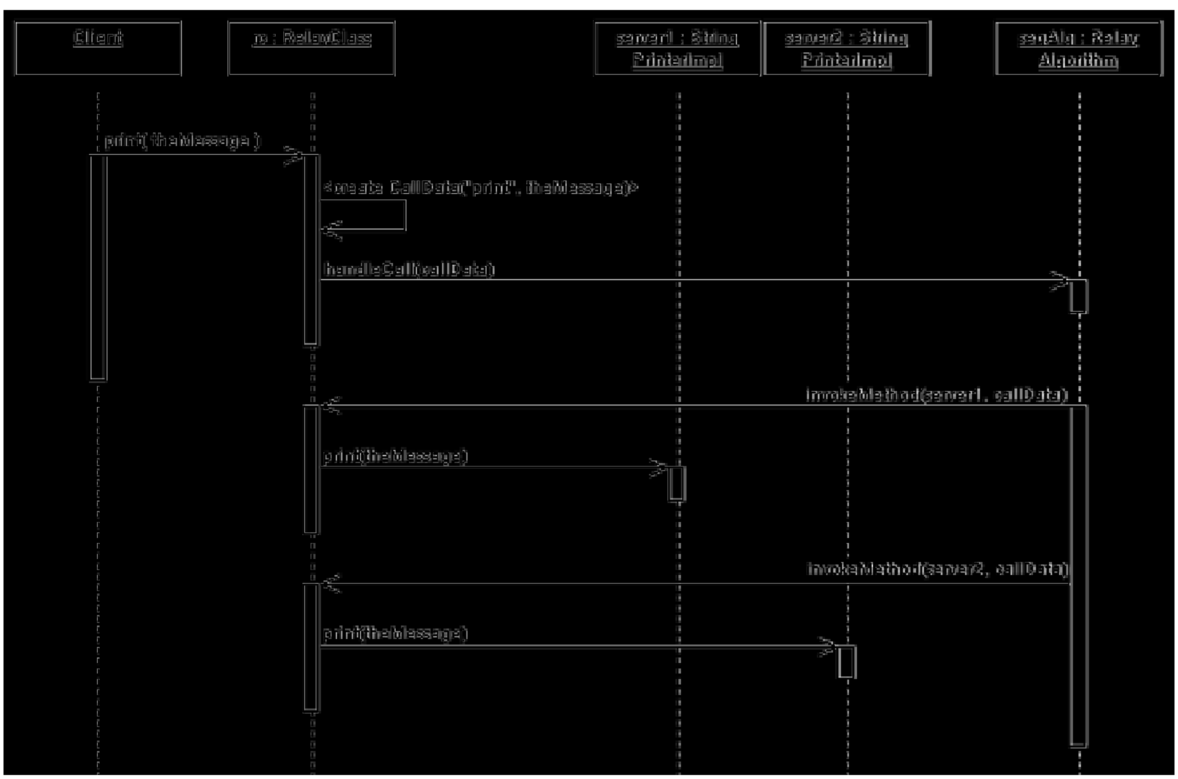

Illustration 3: Method invocation relay as UML sequence diagram

3.2 Algorithms

Algorithms are responsible for handling method invocations. They determine which Server object receives the forwarded method invocation. All algorithms extend the base class RelayAlgorithm to inherit some common behaviour. For example, the class AllDestinationsAlgorithm relays each incoming method invocations to all registered destinations, one after another:

```
  // Java
package de.voelter.odif;

// import statements

public class AllDestinationsAlgorithm
  extends RelayAlgorithm {

    private Vector destinations;

    public void handleCall(CallData cd) {
        for ( int i=0; i<destinations.size(); i++ ) {
            invokeMethod(
              (ODIFBase)destinations.elementAt(i), cd );
        }
    }

    public void init(RelayClass rc) {
        super.init( rc );
        destinations =
          Util.enumToVector( rc.getDestinations() );
    }

}
```

Every algorithm has to override two methods to implement its specific behaviour.

- void init(RelayClass rc) is used to set up the algorithm. It is called after the Relay object has been created. It makes sense to copy the list of destination object so that the algorithm can access it in future calls of handleCall(). All other initilizations have to be performed in this method.
- void handleCall(CallData cd): This is the primary method of the algorithm. It is responsible for relaying incoming method calls. For every incoming call, this method is called once, which can be seen in the StringPrinterRelay example. To invoke methods on a destination object, the invokeMethod() method must be used. This method is implemented by the specific Relay class (see StringPrinterRelay example). The method takes two parameters: The destination object, and the CallData object which has been passed into the handleCall method.

So, creating a new algorithm is usually just a matter of overwriting the handleCall() method. A few examples follow.

Relaying the call to all registered destinations can be achieved as follows:

```
  // Java
public void handleCall(CallData cd) {
        for ( int i=0; i<destinations.size(); i++ ) {
            invokeMethod(
              (ODIFBase)destinations.elementAt(i), cd );
        }
}
```

Relaying to one destination per call, one after another, can be implemented by remembering the position of the last destination in the destinations vector:

```
  // Java
public void handleCall(CallData cd) {
        pos++;
        if ( pos >= destinations.size() ) pos = 0;
        invokeMethod(
         (ODIFBase)destinations.elementAt(pos), cd );
}
```

3.3 Limitations of the current ODIF implementation

Currently, there are a couple of limitations in the ODIF implementation. These limitations are not fundamental in the sense that it was impossible to implement them into ODIF. They have been left out due to time restrictions during ODIF development.

- **No method overloading:** Because ODIF currently identifies the called method only by its name, method overloading is not possible in server classes.
- **Only void methods:** Currently, only methods that use void as their return type can be used. Ways to get around this limitation in the future can be found in the ODIF documentation [ODOC].
- **Code is not moved:** The code (i.e. class files) that is necessary to run the objects on the remote hosts is not moved with the objects. ODIF assumes that the complete code necessary for a system is available on every node.

Illustration 3 shows the relay process in a UML sequence diagram.

4 Tool support

In section 1 we proclaimed that it is desirable to separate the programming and the administration tasks (and roles). Section 3 described the programming side of ODIF, this section focuses on the administrator's task.

ODIF provides a graphical user interface for configuration and system startup. This will be described now.

4.1 The ODIF Server

The ODIF server is the central backbone of every ODIF system. It is responsible for storing the configuration of ODIF systems and to start them. The ODIF Server is a Java application which uses the LPF framework [LPF] to store system configuration data in a relational database such as MS Access or Oracle. The server can be run in startup-only mode or in composer mode:

- **Startup-only mode:** In this mode, the server comes up, starts a system spcified in the server's properties file and terminates. No GUI is involved in this mode. It is just a way to start an ODIF system.
- **Composer mode:** In the composer mode, the server comes up and starts the composer GUI, which can be used to configure systems.

4.2 The Composer

The Composer provides several features, the most important ones will be described briefly. The composer is based on a GUI framework called FAF [FAF].

4.2.1 Browsing the Name Service

To discover which objects are available in the network, the Composer has an integrated Name Service browser. Illustration 4 shows a screenshot.

Illustration 4: Name Service Browser in the Composer

This feature is not directly related to ODIF system configuration. However, because ODIF uses objects located in the Name Service space, it proved practical to be able to browse it.

4.2.2 Working with classes

ODIF needs to know which classes are available in the system. There are three kinds of classes:

- **Server classes:** These classes are implementations of interfaces which do the real work.
- **Relay classes:** These classes are generated from the Server classes and are used to relay method calls.
- **Algorithm classes:** These classes implement the available algorithms.

The Composer allows to administer these classes in the Registered Classes subtree. It provides the following features:

- **Register** the three kinds of classes
- **Verify** that each of these classes is loadable for the server.
- Add textual **information** to describe the classes.
- For Server classes: Once a Server class is registered, it is possible to **automatically create a suitable Relay** class from the Server class. The user has to specify the path where the code should be generated, the name of the Relay class, and some additional information such as whether the newly created Relay class should be compiled and registered with the Composer automatically.

4.2.3 ODIF systems

A system in ODIF is a collection of Server object and Relay objects, together with their configuration, i.e. algorithms and destinations for Relay objects. Furthermore, an ODIF system is structured into nodes and processes.

A *System* consists of n *Nodes* (i.e. computers) connected by a network. On each of these nodes, one or more processes can be run. A *Process* serves as as the environment for ODIF objects. Because all processes are identified by their name and because CORBA provides location transparency, nodes are not a logical element in a ODIF system. A *Process*

Illustration 5: ODIF system hierarchy

running on a *Node* is therefore simply referred to as a *host*.

To create an ODIF system, the following steps have to be carried out:

1. **Create a new system:** A new entry in the System tree node has to be created.

2. **Add hosts:** As each System consists of hosts, the names of the hosts have to be added. This can either happen by selecting them from the current Name Service content, or, if the hosts are not currently available in the Name Service, the names of the hosts can be typed in manually.

3. **Add objects to hosts:** Objects have to be assigned to hosts. To do so, the user has to specify the class name of the new object and the object‘s name in the Name Service once the object is created upon System startup. The objects can of course be Server or Relay objects.

4. **Set algorithms:** For each Relay object, the algorithm that should be used by the Relay has to be set.

5. **Set destinations:** For each Relay object, the destination objects have to be configured. These should implement the same interface as the Relay object.

All this can be done interactively in the Composer. Illustration 6 shows the Composer while the destinations for a Relay object are defined.

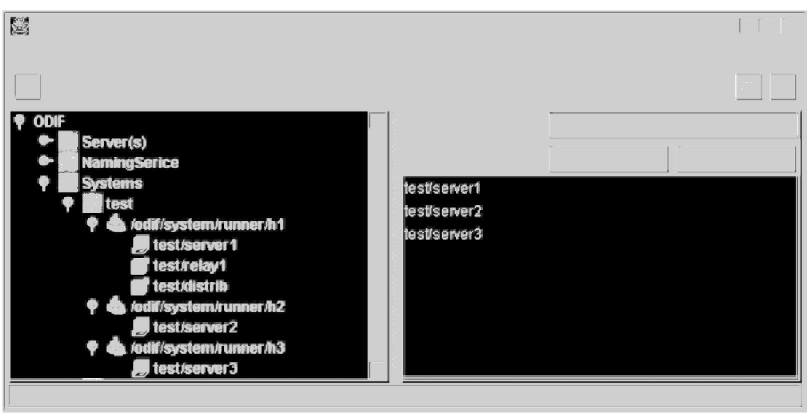

Illustration 6: Defining destinations and the algorithm for a Relay object

The composer can also be used to start an ODIF system interactively. ODIF system startup is as follows:

1. The server verifies, that each of the hosts used in the System is running.

2. It starts the objects configured for each host.

3. It calls the prepare() method on every object. This is the time when algorithms are initialized.

4. The run() method is called on every Server and Relay object.

5 Future improvements

This section describes the enhancements that might be useful for ODIF. It is also a hint as to where future work on ODIF will go.

5.1 Return values

Currently, ODIF only supports method calls with no return values (i.e. void methods). Allowing some form of "return information" is certainly necessary. The proposed solution is based on a variation of the Asynchronous Completion Token pattern by Doug Schmidt et. al. [ACT], which uses tokens remembered by clients and servers to identify the asynchronously returned values for method invocations. An evaluation of the different approaches to solving the non-void return value problem can be found in the ODIF documentation [ODOC].

5.2 Use of XML

When a Relay object is started, it is passed some startup information: its destinations, its algorithm, etc. To create a general format for such information, the use of XML seems useful.

5.3 Configurable Algorithms

An algorithm might have some configuration options, e.g. the number of times it should retry to contact a specific server, some timeout, or a queue length. Currently there is no way to parametrize an algorithm with such information when the system is configured.

To allow for algorithm configuration, two ways seem to make sense:

- Use of Pluggable Component: An algorithm object could be implemented as a Pluggable Component [PC]. When the Algorithm is deserialized in the using Relay object, it is already configured.

- Query for Parameter Names: Each algorithm might have a method called Enumeration getParameterNames() which returns a list of parameter names. When an algorithm is configured, the algorithm is queried for the parameter names and the user can then specify values for them. When the Relay object is started, the values for its parameters are supplied via the startup XML string (see 5.2).

In any case, the administrator must be able to specify these configuration options, which means that a GUI must be available which has to be shown in the Composer when the Algorithm is set.

5.4 System launch

System launch is currently quite primitive. First, all Servers are started, then all Relays. Clients are not included in the startup process at all, and there is no way to start components that enter the system after initial startup, etc. The use of techniques such as those used in DASA's CORBAtschoff ([DASACORBA]) might make sense. There, it is possible to add Servers while the system is running. They are initialized properly, etc. CORBAtschoff was developed by Markus Voelter and Oliver Kellogg at DASA, Ulm.

5.5 AgentODIF

ODIF is about distribution. It launches objects on remote hosts and then uses the services of these objects. Currently, ODIF does not move the code needed by the objects to the remote hosts. This is definitely a point for improvement. Agent technology can help there. Agents are mobile objects which move their state *and their code* to remote hosts. For ODIF, this could bring the following benefits:

- Relay and Server objects (then agents) could be created locally. Then they are passivated, they "sleep" on the local host. Upon system startup, the agents are transported to their destination host, where they are activated and can be used like any other CORBA object.
- Using agents, or more specifically, IBM's Aglets API ([AGWEB], [AGBOOK]), would not change anything in ODIF's distribution concepts (Relays, algorithms), only the way how the Server and Relay objects are distributed would change.

Experimenting with Agent technology in the context of ODIF would certainly be very interesting.

6 Related work and products

Load distribution and redundancy is not a new requirement nor is ODIF the first tool that implements these features. The following sections describe which other tools are available on the market today and how they compare to ODIF.

6.1 Available tools

There are many tools available that implement ODIF's features in object environments. Two kinds will be described here.

6.1.1 Object Transaction Monitors

Transaction processing monitors are among the most important types of applications today. They provide load balancing, redundancy, etc. for transaction servers. OTMs (Object Transaction Monitors) provide the same for transactional objects. Based on OMG's Object Transaction Service, they provide a runtime environment that handles the following features: load balancing, redundancy, logging, security, replication, and queueing.

All these features are necessary to allow for very large OLTP (OnLine Transaction Processing) applications like for example a world wide airline booking system. TP monitors have been in widespread use for decades, and new applications are developed using their object-oriented cousins, the Object Transaction Monitors (OTMs). Vendors also provide tools to integrate new OTM applications with applications written for traditional TP monitors. BEA's M3 is an example, it is built on BEA's TUXEDO TP monitor.

However, OTMs have one disadvantage: Usually these tools are really large, complex systems that need qualified administration staff and a powerful IT infrastructure. For smaller applications that would like to benefit from the features described above, OTMs are overkill and too expensive.

6.1.2 Components, EJBs and Application Servers

A new trend in distributed object systems are components, or better, CBD, Component Based Development. One of the most promising component architectures are EJBs, the abbreviation stands for Enterprise Java Beans. There, all components live in a container, too. To allow this container to manage arbitrary components, all components must be programmed according to some conventions specified in the definition of the EJB component model. The container which hosts components is usually called an EJB Application Server in the case of Enterprise Java Beans. The market is currently full of this kind of applications, every major (and some minor) software company has an application server in its portfolio. The range of features is very diverse, as it might be expected in such a (too) fast growing market. However, these containers provide many of the same features as OTMs but are considerably easier to program. Requiring an application server and a great deal of component-model specific programming , these components are still not what one would call a lightweight approach to load balancing, redundancy, failover, replication, etc.

6.2 ODIF is different

Comparing ODIF to the tools and systems described above reveals one big difference: ODIF is small and lightweight. OTMs and EJBs all require a rich infrastructure and very strict programming guidelines. ODIF only uses standard CORBA features and provides only one tool, the Composer. By separating what should be distributed from the „how“, it is nonetheless useful.

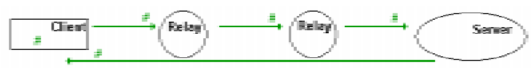

Illustration 7: The solution for the return value problem is based on the Asynchronous Completion Token [ACT].

7 Summary

The development and prototype implementation of ODIF has shown that the concept of Servers, Relays and algorithms works. The major advantages have been verified and the approach has been compared to other approaches in the current field.

The ODIF prototype is by no means complete, nor is it a production-ready system, but it works fine and shows that ODIFs concepts can be brought to practical use.

A minor goal was to show how frameworks can speed up software development. That's the reason why parts of the ODIF system have been developed s separate frameworks. These two frameworks, LPF and FAF, have not only been used in ODIF, both have been used in another application by voelterSOFTWARE and in a large project-management software by Zimmermann Business Solutions. In all three projects, the development process could be speeded up significantly and the initial bug rate was significantly lower because of the use of frameworks.

8 Reference

[ODOC] Markus Voelter, *ODIF Documentation*, Diploma thesis, Fachhochschule Ravensburg-Weingarten or http://www.voelter.de

[AGWEB] IBM, *Aglets Home Page*, http://www.trl.ibm.co.jp/aglets

[AGBOOK] B. Lange, Mitsuru Oshima, *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley 1998, ISBN 0-201-32582-9

[ACT] Irfan Pyarali, Tim Harrison, Douglas C. Schmidt, *Asynchronous Completion Token*, Pattern Languages of Program Design 3 (Chapter 14) Addison-Wesley 1998, ISBN 0-201-31011-2

[LPF] Markus Voelter, *Little Persistence Framework*, Diploma thesis, Fachhochschule Ravensburg-Weingarten or http://www.voelter.de

[FAF] Markus Voelter, *Flexible Application Framework*, Diploma thesis, Fachhochschule Ravensburg-Weingarten or http://www.voelter.de

[PC] Markus Voelter, *Pluggable Component*, Proceedings of the EuroPLOP 99 conference or http://www.voelter.de

[DASACORBA] Markus Voelter, Oliver Kellogg, *CORBAtschoff*, industry semester report, Markus Voelter, DASA Ulm