

Introduction to ORBLink

ORBLink is a CORBA Object Request Broker (ORB) for Allegro Common Lisp. ORBLink allows a Lisp program to communicate with programs written in other languages, possibly running on other machines.

CORBA stands for Common Object Request Broker Architecture. CORBA is a set of protocols that enable distributed applications, potentially running in multiple languages, to communicate with one another. The CORBA specification is managed by the **Object Management Group**, at <http://www.omg.org>, a consortium of hundreds of industry, government, and academic institutions. Another good set of links is available from Cetus.

CORBA data types are specified in a simple description language called **IDL**, the Interface Description Language.

ORBLink provides functionality that converts IDL into native Lisp types, and that allows a Lisp program to invoke operations on objects, possibly remote specified in IDL.

There are numerous books and references on CORBA. In addition, here is another high-level, non-technical overview of ORBLink based on a cover story on the Franz Inc. Web site.

Because CORBA applications typically involve multiple distributed components, they can become somewhat complicated. Feel free to ask questions of orbink@franz.com.

CORBA-Enabling Allegro CL Applications

This document is a slightly edited version of an introductory article on Allegro ORBLink that first appeared as a cover story on the Franz Inc. Web site in June, 1998. It is intended as non-technical overview of CORBA and ORBLink. For specific technical information, see the main documentation supplied with ORBLink itself. With Allegro ORBLink, Allegro CL applications can:

- Invoke methods on objects on remote machines
- Receive remote invocation requests from other processes
- Interoperate with objects written in Java, C++, C, COBOL, Smalltalk and other CORBA-compliant languages
- Access common CORBA services
- Interoperate with other commercial ORBs such as ORBIX and Visibroker

So what is CORBA and How Does it Work?

The Common Object Request Broker Architecture (CORBA) is a standards-based distributed computing model for object-oriented applications developed by the Object Management Group (OMG), a group of 700 vendor and user members including HP, Novell, Sun, IBM, and Digital. Franz Inc. is also a member of the OMG, and, using Allegro ORBLink, fully supports the CORBA architecture for distributed objects.

CORBA allows objects executing on one system to call methods on objects executing on another system. Finding the objects, transmitting the method call and returning any results are mediated using an Object Request Broker (ORB) that the application doesn't have to know about. The ORB allows objects to interact in a heterogeneous, distributed environment, independent of the computer platforms on which the various objects reside and the languages used to implement them. For example, a C++ object running on one machine can communicate with an object on another machine that is implemented in Lisp.

CORBA-compliant applications typically comprise a *client* and a *server*. The client invokes operations on objects that are managed by the server, and the server receives invocations on the objects it manages and replies to these requests. For example, a Lisp object can either use CORBA services available over the network (as a client), or it can publish services to other components in the application (as a server). The ORB manages the communications between client and server using the Internet Inter-ORB Protocol (IIOP), which is a protocol layer above TCP/IP.

The objects in a distributed CORBA application, regardless of the language in which they are implemented, can communicate with the ORB through an interface written in the CORBA Interface Definition Language (IDL). The CORBA IDL is a language included in the CORBA 2.0 specification that is designed to describe the interfaces of objects, including the operations that may be performed on the objects and the parameters of those operations. The behavior of an object is thus captured in the interface independently of the object's implementation. Clients need only know an object's interface in order to make requests. Servers respond to requests made on those interfaces, and clients do not need to know the actual details of the server's implementation.

To implement an interface, CORBA IDL is compiled into the source code language with which the client or server is implemented. On the client side, this code is called a *stub*. On the server-side, this IDL code is called a *servant*.

In order to request a service from the server, the client application calls the methods in the stub (moving down the protocol stack from the client to the stub to the ORB). Requests are then handled by the ORB, and enter the server via the servant (moving up the protocol stack: i.e. an *upcall*). The server object then services the request and returns the result to the client.

CORBA Illustration 1

Third-party ORBs such as ORBIX from IONA Technologies and Visibroker from Visigenic/Borland provide common CORBA services such as name services and transaction services. A name service allows a client to initiate contact with a remote object. The client asks the name service for a connection to the remote object by name. The name service works with the stub code to return a connection, permitting the client to call methods on the remote object. This work by the name server might include actually starting up the server that implements the remote object.

Writing IDL stubs and servants for Allegro CL applications enabling them to communicate with other distributed CORBA-compliant objects is made possible by Allegro ORBLink. Users do not need to know low-level details such as the IIOP protocol; Allegro ORBLink handles the mapping for them so that they can write regular CLOS code to make their applications fully CORBA-ready and interoperable with other third-party ORBs.

Allegro ORBLink: Adding CORBA-Compliance to Allegro CL

Allegro ORBLink is an add-on product for Allegro CL that comprises an automatic mapping of the CORBA IDL into Allegro CL. This mapping is fully IIOP-compliant, and it permits Allegro CL users to write CLOS code that takes advantage of CORBA services or that publishes CLOS services to other application components as described above. Fully compatible with other third-party ORBs, Allegro ORBLink enables Allegro CL applications to take advantage of common CORBA services such as name and transaction services.

Features of Allegro ORBLink include:

- Complete IIOP compliance
- Automatic exception handling
- Support for all IDL data types
- Open support for industry standards on a variety of hardware and operating system platforms
- Integration with existing systems written in any other CORBA-compliant programming language
- Interoperability with other CORBA-compliant objects and services
- Full compatibility with common object services defined in CORBA, including naming, events, and transaction services available through third-party ORBs such as Orbix from Iona and Visibroker from Inprise.

An Example CORBA-Compliant Distributed Application

The following diagram illustrates a simple client-server banking application with a server component written in Common Lisp and a client (say an ATM terminal) written in Java. The file containing the CORBA interface definitions is in this example called "bank.idl". Here is the IDL source code in this file:

```

module Bank(
interface Account {
    float Balance();
};
interface AccountManager{
    Account open (in string name);
};
};

```

The "bank.idl" file is compiled into two pieces: it is compiled by a Java CORBA compiler into a Java stub on the client side, and it is compiled by Allegro ORBLink into a CL servant on the server side.

CORBA Illustration 2

The classes *Bank:Account* and *Bank:AccountManager* constitute the CL servant. The servant also includes methods on these classes which deal with marshalling, events, IIOP, etc.

The user code for the servant extends the servant classes and implements the methods using *define-method*.

```

(defclass Account (Bank:Account)
  (balance :accessor get-balance))

(corba:define-method balance ((account Account))
  (get-balance account))

(defclass AccountManager (Bank:AccountManager)
  ((accounts :accessor get-accounts
    :initform (make-hash-table :test #' equal))))

(corba:define-method open ((mgr AccountManager) name)
  (gethash name (get-accounts mgr)))

```

The *AccountManager* class has an *open* method that takes an account holder's name and returns the account for that account holder. The *Account* class has a *balance* method that takes an Account object and returns a balance.

corba:Define-method uses all of the "hidden" servant methods automatically where needed (for marshalling, IIOP, etc.). The programmer does not need to manually invoke them. As such, Allegro ORBLink makes it easy to write servants and stubs with no requirement to manually handle details of the underlying protocols.

To use an account as a client, one would write code such as:

```

(setq JS (op:open Manager "John Smith")) => obtains the account in the name of John Smith
(op:balance JS) => returns the balance in the JS account

```

CORBA References

For more information about CORBA, visit the OMG website at <http://www.omg.org>, which includes all of the official CORBA standards. This site, particularly the page <http://www.omg.org/news/begin.htm>, contains a wealth of information on learning about CORBA. Another excellent site is Cetus. There are also a number of books on CORBA. . If you are interfacing with Java - and to some extent even if you are

not - we recommend books such as:

- Programming with Visibroker. Doug Pedrick, Jonathan Weedon, Jon Goldberg, and Erik Bliefeld. Wiley, 1998
- Java Programming with CORBA by Andreas Vogel and Keith Duddy, 2nd edition, John Wiley & Sons, February 1998.
- Client/Server Programming with Java and CORBA, Robert Orfali and Dan Harkey, 2nd edition , John Wiley & Sons, February 1998.
- Programming with Java IDL Geoffrey Lewis, Steven Barber, and Ellen Siegel. John Wiley & Sons, November 1997.

There are also a number of books available on CORBA without reference to any language. All of these books are easily accessible and may be ordered from <http://www.amazon.com>. However, the most important CORBA document is always the latest OMG core CORBA specification.

Installing ORBLink

In order to install configure ORBLink, it is necessary to modify one file:

orblink-configure.cl

This file is located in the `code` subdirectory of the ACL home directory - the same directory in which **orblink.fasl** is located. ORBLink will run out-of-the box with no installation steps. However, in order to generate object references that can be used by programs in other internet domains, you need to apprise ORBLink of the internet domain to use in these object references. In most cases, it is sufficient to complete the installation by modifying the string indicating the domain in **orblink-configure.cl**:

```
(setf
  (op:domain corba:orb)
  "" ; This may be modified to the actual domain,
    ; for example, "franz.com"
)
```

However, sometimes the default rules used by ORBLink to generate a fully-qualified hostname from a domain name do not work. In this case, you should set the IP of the host on which ORBLink is to be run directly as described in **orblink-configure.cl**:

```
(setf (op:host corba:orb)
  "" ; Replace with, e.g., "192.0.0.2"
    ; for the Internet address or fully-qualified
    ; IP address of the given host
)
```

By default these lines are commented out. A natural question is: how can I be sure if the default method suffices? The simplest way: from a Lisp listener evaluate:

```
(require :orblink)
corba:orb
```

This should print a description of the ORB, for example:

```
ORBLink:ORB Allegro ORBLink :version 1.0 :host "corba.franz.com" :user "stiller"
```

The "host" field here should denote the fully qualified hostname of the host on which ORBLink is to be run. If it does not so denote, then you should set the host explicitly. This method also provides a simple check that the installation is successful.

Mapping IDL to Common Lisp

The Common Lisp to IDL mapping document defines the official proposed mapping of Common Lisp to CORBA. Franz Inc. is working with OMG to standardize this mapping. The mapping document is currently provided in **.pdf** format. If you are unable to read a **.pdf** file, you should do one of the following:

1. Download and install the Acrobat Portable Document Format reader:

[IMAGE]

2. Contact Franz Inc. for a hardcopy of the documentation.

The mapping document is located in the file **mapping.pdf** of the **doc** subdirectory of the ORBLink root directory.

Summary of the IDL/Common Lisp Mapping

This document summarizes highlights and key features of the IDL to Common Lisp mapping for the ORBLink programmer. Each IDL type is mapped to a corresponding Lisp type:

- Primitive types
- Sequences
- Interfaces
- Operations
- Structs
- Unions
- Exceptions
- Typedefs and const definitions
- Understanding the type rules

Mapping for primitive types

The mapping for primitive types is straightforward:

- IDL boolean maps to Lisp boolean
- IDL short, long, octet, unsigned short and unsigned long map to Lisp sub-types of integer containing the appropriate integers. For example, the lisp type `corba:short` describes 16-bit Lisp integers.
- string maps to Lisp string; char maps to Lisp character.
- arrays maps to Lisp array of the same rank.

As a practical matter, this means that you can pass a value of the expected Lisp primitive type as a parameter to any IDL operation declared to accept that type. We will see some more examples below.

Mapping for sequences

An IDL sequence is mapped to a Lisp sequence, that is, to a vector or to a list.

This can cause confusion: when is a vector allowed or required? when is a list allowed or required?

The general rule is: if an IDL operation expects a parameter that is a sequence, either a vector or a list may be passed to that operation.

If an IDL operation returns a sequence parameter, it may be either a vector or a list. ORBLink guarantees that if the invocation was remote then the returned sequence value will be a vector, not a list.

Mapping for interfaces

The most important mapping to know is that of interfaces.

Consider the following interface named I defined in module M:

```
module M {  
  interface I { attribute long a1;  
                readonly attribute string a2;}  
}
```

When compiled via `corba:idl` three classes are created in the Lisp:

1. A Lisp class named M:I (that is, the symbol named "I" in package M).
2. A Lisp class named M:I-servant, inheriting from M:I and from `corba:servant`
3. A Lisp class named M:I-proxy, inheriting from M:I and from `corba:proxy`.

Instances of M:I-proxy are proxies: methods invoked on them are marshalled and forwarded to remote implementations. Instances of M:I-servant are servants: these are normal Lisp classes that implement various methods. They can receive remote invocations. When a method is invoked on a servant no marshalling is done: the servant implementation code is invoked directly. Now suppose an interface named J extends I:

```
module M { //We are reopening the module M  
  interface K {;}  
  interface J (I, K) {;}  
}
```

When compiled, the Lisp classes corresponding to J will have the following inheritance structure:

1. M:J inherits from M:I and M:K
2. M:J-servant inherits from M:I-servant and M:J-servant
3. M:J-proxy inherits from M:I-proxy and M:K-proxy

Attributes are mapped to slots in the generated -servant class. For an attribute a:

1. The slotname is `op:a`
2. The reader name is `op:a`
3. The writer name is `(setf op:a)` - unless the attribute is readonly, in which case no writer is generated.
4. The initarg is `:a`

For example, M:J-servant inherits a slot named `op:a1` from M:I-servant. An instance of M:J-servant can be created via

```
(setq j-serv (make-instance 'M:J-servant :a1 3958810))
```

The value of the attribute `a1` of `j-serv` can be retrieved via:

```
(op:a1 j-serv)
----> 3958810
```

The value of the attribute can be changed via:

```
(setf (op:a1 j-serv) -3)
(op:a1 j-serv)
---> -3
```

The readers and writers can also be invoked on proxies. However, the `initarg` cannot be used for a proxy.

Mapping for operations

The mapping for operations is quite simple: if `foo` is an operation defined in some interface, then there is a corresponding Lisp method named `op:foo`. If `foo` is declared to return `void` and has no out arguments, then `op:foo` returns no values.

If `foo` is not declared to return `void` and has no out arguments, it returns a single value.

If `foo` is declared to return out arguments then they are returned as multiple values after whatever values `foo` would have returned without those out arguments.

The first argument to `op:foo` is the object on which it is invoked. The remaining arguments are the parameters of the operation. Consider the following IDL:

```
module M {
  interface optest {
    void testvoid (in long a);
    long testlong (in string b);
    long testmultiple (in string b, out char c);
    void testvoidmultiple (out char c1, out string s, out float f3);
  };
};
```

Suppose that `m` is an object of type `M:optest`. Then an invocation sequence might look like:

```
(op:testvoid m -100000)
--->[No values returned]

(op:testlong m "hello")
--->599934

(op:testmultiple m "goodbye")
---> 3000 #\G

(multiple-value-bind (a b c) ;Get the returned values
  (op:testvoidmultiple m)
  (list "The values returned are:" a b c))
---> ("The values returned are" #\Y "hurdle" 1.87)
```

Everything that "looks like a method" in Lisp is mapped to the `op` package, notably struct member accessors, union member accessors, attribute accessors, and so on.

structs

A struct follows the same naming rules as an interface. A struct named `S` defined in module `M` maps to a Lisp class named `M:S`. If `S` has a member named `mem`, then the Lisp class has a slot named `op:mem` with `initarg :mem` and accessor `op:mem`. The constructor for a struct has the same name as the struct. For example from the IDL:

```
module M {  
  struct S {long foo; string fum;};};
```

You can create a new instance of the class `M:S` via

```
(setq struct (m:s :foo 300 :fum "test"))  
----> #< M:S :FOO 300 :FUM "TEST">
```

```
(op:foo struct)  
----> 300
```

```
(setf (op:fum struct) "passed")  
----> "passed"
```

```
(op:fum struct)  
----> "passed"
```

unions

A union follows the same naming rules as interfaces and structs. The member accessors and member `initargs` follow the usual pattern, except that of course only one `initarg` can be used in the initialization (since only one member can have a value). A union has two slots named `op:union-value` and `op:union-discriminator`. The name of the constructor for the union is the name of the union. Consider for example the following IDL:

```
module M { interface IU  
  { union V switch (long) {  
    case 3: string foo;  
    case 5: long fum;};};};
```

The name of the corresponding Lisp class is `M:IU/V`. (The `/` is a scoping separator). You can create a union with label 3 and value "echo" in the `foo` member via:

```
(setq u (M:IU/V :foo "echo"))
```

The value can be retrieved via:

```
(op:foo u)  
----> "echo"
```

```
(op:union-value u)  
----> "echo"
```

```
(op:union-discriminator u)  
----> 3
```

Exceptions

The mapping for exceptions is exactly like the mapping for structs, except that an IDL user-defined exception is a subclass of the Lisp condition class `CORBA:userexception`, which inherits from `CORBA:exception`, which inherits from `condition`.

Typedef and const definitions

You can define a named type or a named constant in IDL. As would be expected, these are mapped to Lisp types and constants respectively, following the usual naming rules:

```
module M {
  typedef string foo;
  const long r = 1 + 3;
};
```

From the above IDL, we get:

```
(typep "hello" 'M:foo)
---->T
(typep 3 'M:foo)
---->nil
M:r
---->4
```

Understanding the rules for constructed types

The mapping rules were constructed with the primary goal of uniformity and ease of learning.

The fundamental abstraction used is that of a "type" with some kind of "named members". Insofar as possible, in each case such a type maps to a Lisp type of the same name, with a keyword `initarg` corresponding to each named member, and readers and writers for each named member. Each reader is in the `op:` package; the writer is formed via `(setf reader)`. The constructor is simply the name of the type.

Lexical conventions

The ORBLink documentation necessarily involves simultaneous discussion of IDL datatypes and their associated Lisp types. These types typically have names that are quite similar.

IDL is also case-sensitive, whereas ANSI Common Lisp is not.

Unfortunately it is the case that in the current version of this documentation there is no standard font or case distinction that would allow the reader to distinguish lexically the name of an IDL type from the name of the corresponding Lisp type.

In most cases the context should make the distinction clear. When it does not, please contact orbblink@franz.com.

Getting started with ORBLink: A tutorial

Prerequisites for working through the tutorial

These are the prerequisites for running the tutorial:

1. You should be able to load Allegro Common Lisp and ORBLink.
2. You should have some familiarity with CORBA
3. You should have read the IDL/Lisp mapping document.
4. You should know where the ORBLink home directory is located.

However, even if you do not have all the prerequisites, you might find the tutorial useful to get an idea of what an ORBLink application entails.

Because of the length of this tutorial, you may find it more convenient to print out rather than to read on-line.

Overview of the tutorial

In this tutorial, we will work through a very simple example of implementing IDL in Lisp and then invoking the Lisp client. After doing this, we will see how to modify the server itself on the fly.

We will assume that there are two distinct Lisp worlds that have been started: one is called the *server* and the other the *client*.

Commands that should be typed into the server or client listener are prefaced with `[server-listener]USER>` or `[client-listener]USER>` in the text below.

In broad terms, to start our application we will:

- start a server,
- start a client,
- invoke methods from the client to the server.

Our tasks are thus logically partitioned into two categories: *server-side* and *client-side*.

Server side steps

1. Load ORBLink
2. Compile the IDL on the server
3. Define the implementation classes and methods
4. Instantiate a server object
5. Publish the IOR of the server object

Client side steps

1. Load ORBLink on the client
2. Compile the IDL on the client
3. Retrieve the IOR of the server object and generate a client proxy from this IOR
4. Invoke methods on the client proxy.

These method invocations will result in the forwarding of remote requests to the server object.

It is important to note that either the client or the server side can be implemented in any CORBA-compliant language; that the client or server process can reside on the same or on a different computer in numerous platforms; and that the overall structure of these steps is the same for any language/platform configuration.

After these basic steps we will show how a Lisp server or client can be modified dynamically.

Let us, thus, begin by starting the server side:

Loading ORBLink on the server

The Lisp source code for this example is included in the directory `examples/grid/cl/`. You can load the example code from any directory. However, for specificity we will assume here that the current working directory is the root of the ORBLink home installation. You can use the `:cd` from within a Lisp listener to set the current working directory of the listener, e.g.

```
[server-listener]USER> :cd /usr/acl-5/code/orblink
[server-listener]USER> (require :orblink)
```

We recommend as well setting the current package to `user`:

```
:package :user
```

Compile the IDL on the server

The IDL source code for this example encapsulates the interface to a simple two-dimensional array of strings:

```
module example {
  interface grid;
  interface grid {
    readonly attribute short height;
    readonly attribute short width;
    void set(
      in short n,
      in short m,
      in string value
    );
    string get(
      in short n,
```

```

        in short m
    );
};
};

```

To compile the IDL, give the pathname of the IDL as the argument to the IDL compiler:

```
[server-listener]USER> (corba:idl "examples/grid/idl/grid.idl")
```

This will define the classes `example:grid`, `example:grid-proxy`, and `example:grid-servant`. The `corba:idl` function will return an Interface Repository object that encapsulates in CORBA compliant format the definitions in the IDL file.

The class `example:grid-servant` already has defined slots named `op:width` and `op:height`, corresponding to the attributes in the IDL definition of the `grid` interface. These slots have pre-defined readers named `op:width` and `op:height` with corresponding initialization arguments `:width` and `:height`.

Define the implementation classes and methods

In order to write a server for the `grid` interface, it is first necessary to define a class that extends `example:grid-servant` (technically this step is unnecessary, insofar as the new methods could be defined on the `example:grid-servant` class directly, but this usage would be poor style).

We will name our user-defined class, which extends `example:grid-servant`, `user::grid-implementation`.

Our class `user::grid-implementation` will extend `example:grid-servant` and will include a single extra slot named `array` that holds the actual values in the grid.

The `example:grid-servant` class defines slots named `op:width` and `op:height` and, since these are readonly attributes, it defines readers of the same name. Our `grid-implementation` will add default initforms to these slots of values 4 and 5 respectively. Users who wish to initialize the grid with a different size can do so using the automatically defined `:width` and `:height` initargs.

Our class definition thus looks like this (or see the file `examples/grid/cl/grid-implementation.cl`):

```
(defclass grid-implementation (example:grid-servant)
  (
    (op:width :initform 4)
    (op:height :initform 5)
    (array)))
```

We define an initializer for this class that simply initializes the `array` to the size specified by the values of the slots named `op:width` and `op:height`, with initial element the string `"initial"`:

```
(defmethod initialize-instance :after ((this grid-implementation) &rest args)
  (setf (slot-value this 'array)
        (make-array `((, (op:width this) ,(op:height this)) :initial-element "Initial"))))
```

Finally, we implement the IDL operations named `get` and `set`. Because these are IDL operations, their implementation must be via the `corba:define-method` macro. The syntax of `corba:define-method` is specified as part of the CORBA IDL/Lisp mapping and closely follows the syntax of the usual `defmethod` macro.

For example the `get` method is implemented as:

```
(corba:define-method get ((this grid-implementation) row column)
  (aref (slot-value this 'array) row column))
```

You should now load the file that contains the definitions of the grid implementation class and its associated methods:

```
[server-listener]USER> :ld examples/grid/cl/grid-implementation.cl
```

Instantiate a server object

We can now verify that the appropriate classes have been loaded by instantiating an instance of `user::grid-implementation`:

```
[server-listener]USER> (setq test-grid (make-instance 'grid-implementation))
[server-listener]USER> (op:set test-grid 1 2 "This is a test.")
[server-listener]USER> (op:get test-grid 1 2)
--> "This is a test"
```

(Note that not all responses from the Lisp listeners are printed here).

The ORB itself is only involved implicitly in this computation; there is no marshalling or unmarshalling, and no socket connections, involved. The `op:get` and `op:set` methods are normal CLOS methods.

Publishing the IOR

In order to invoke methods on `test-grid` remotely, it is necessary to publish the IOR (interoperable object reference) of `test-grid`.

The IOR of the `test-grid` object can be obtained by invoking the `op:object_to_string` method on the ORB itself. The ORB is always bound to `corba:orb`:

```
USER> (op:object_to_string corba:orb test-grid)
--> [a long string of characters beginning with "IOR"]
```

As a side effect of computing this string, called a *stringified IOR*, a TCP socket listener has been started that waits for invocations on the `test-grid` object. Since no client yet knows the IOR of `test-grid`, however, no invocations can be forthcoming until we make this IOR available to a client.

The simplest way to make the IOR available is for the server to write the IOR to a file that is then read by the client. This method is not particularly general, of course, but it will suffice to run simple examples. Choose a file for storing the IOR that is both writeable by the server and that can be read by any client. For example, you can try using the filename `[directory]/grid.ior` where the string "`[directory]`" in the following should be replaced by some directory to which you have write access:

```
USER> (orblink:write-ior-to-file test-grid "[directory]/grid.ior")
```

You should verify now that the IOR string you computed above has indeed been written to the file `[directory]/grid.ior`. Note that you can examine the source to the `write-ior-to-file` function in the file `examples/ior-io/cl/sample-ior-io.cl`:

```
(defun orblink:write-ior-to-file (object pathname)
  "Writes the IOR of object, the first argument, to the file denoted by pathname, the
  second argument. Because this routine is primary explanatory, little error checking is
  performed. If *default-ior-directory* is non-nil, pathname is first merged with
  *default-ior-directory*"
  (when *default-ior-directory*
    (setq pathname (merge-pathnames pathname *default-ior-directory*)))
  (ensure-directories-exist pathname) ; Create intermediate directories if necessary
  (with-open-file
    (stream pathname :direction :output :if-exists :supersede)
    (format stream ("~A" (op:object_to_string corba:orb object)))
    (format t "Wrote ior to file: ~a~%" pathname)
    (force-output)
    t))
```

Starting the client: Load ORBLink and compile the IDL on the client

Now that the IOR of the `test-grid` object has been published in a "well-known" place, a client can bind to it. You should start a new Lisp world for this portion of the tutorial, perhaps on a different machine, and then restart ORBLink and recompile the file `examples/grid/idl/grid.idl`. Thus there are now two Lisp listeners: the client and the server. (In fact this example will work just as well if the client and server are implemented in the same image and the same listener, but it is clearer for the exposition to separate them).

```
[client-listener]USER> (require :orblink)
[client-listener]USER> (corba:idl "examples/grid/idl/grid.idl")
```

Generate a client proxy for the server object

The process of generating a proxy is conceptually divided into two phases: reading the IOR and converting the IOR into a proxy.

Since the IOR now resides in the file `[directory]/grid.ior`, it may be read simply via:

```
[client-listener]USER> (setq ior
                        (with-open-file (stream "[directory]/grid.ior" :direction :input)
                          (read-line stream)))
```

This form should return, in the client listener, the same long string that was returned above in the server listener as the result of calling `op:object_to_string` on the `test-grid` object.

The next step is to create a proxy from this IOR. This can be done using the CORBA compliant `string_to_object` operation on the ORB:

```
[client-listener] USER> (setq test-grid-proxy (op:string_to_object corba:orb ior))
```

This should return an instance of type `example:grid-proxy`. This proxy may then be used to invoke operation on the server-side object in the server image from the client image.

Note: the preceding two steps, reading an IOR from a file and forming a proxy from that IOR could have been coalesced into the single invocation:

```
(setq test-grid-proxy (orblink:read-ior-from-file "[directory]/grid.ior))
```

The source code for the `orblink:read-ior-from-file` function is also located in the file `examples/ior-io/cl/sample-ior-io.cl`.

Invoke methods on the client proxy

You can now invoke methods on the `test-grid-proxy` object using exactly the same calling sequence as you did to invoke methods directly on the `test-grid` object:

```
[client-listener]USER> (op:set test-grid-proxy 1 3 "proxy-test")
[client-listener]USER> (op:get test-grid-proxy 1 3)
---> "proxy-test"
```

You can verify from the server world that these values really have changed:

```
[server-listener]USER> (op:get test-grid 1 3)
----> "proxy-test"
```

This concludes the first part of the tutorial.

Next, we discuss the issue of modifying the server on the fly.

Modifying the server

One convenient feature of Lisp is its ability to add functionality to the server without stopping and restarting the application. To demonstrate this functionality, we presume that it is desired to augment the `grid` object with a new attribute, say `name`. Make a copy of the file `examples/grid/idl/grid.idl` and modify the copy by adding the line

```
attribute string name;
```

in the interface definition. Now recompile the modified file using the `corba:idl` function.

For your convenience the modified version is in the file `examples/grid/idl/grid-modified.idl` and thus the recompilation step is done via:

```
[server-listener]USER> (corba:idl "examples/grid/idl/grid-modified.idl")
```

If you now evaluate the form `(describe test-grid)` on the server side, you will see that a new slot named `name` has indeed been added to the `test-grid` object. Now let's set the value of this new slot:

```
[server-listener]USER> (setf (op:name test-grid) "Modified grid")
[server-listener]USER> (op:name test-grid)
----> "Modified grid"
```

Modifying the client

In order for the client to invoke the newly defined methods on the `test-grid` proxy, it also needs to recompile the IDL source:

```
[client-listener]USER> (corba:idl "examples/grid/idl/grid-modified.idl")
```

Now the client can invoke the new methods:

```
[client-listener]USER> (op:name test-grid-proxy)
----> "Modified grid"
[client-listener]USER> (setf (op:name test-grid-proxy) "client-modified name")
[client-listener]USER> (op:name test-grid-proxy)
----> "client-modified name"
```

Moving on

ORBLink offers many features not discussed in this introductory tutorial, among which are customizable exception handling, handling many other data types, support for persistent IORs, any handling, and so forth.

Parting words on the tutorial

CORBA is not always as simple as in this case. In general, there will be configuration problems in starting up different ORBs: all sorts of environment variables have to be set up correctly and various daemons need to be started. Once these configuration issues are resolved, the actual invocation of methods on remote objects is normally straightforward.

Using the IDL Administrative interfaces

ORBLink ORB administrative interfaces are themselves specified in IDL, or more technically in *pseudo-IDL*. Pseudo-IDL is like IDL except that:

1. A pseudo-interface, an interface specified in pseudo-IDL, does not generate `-servant` or `-stub` classes, and
2. a pseudo-object may not be passed remotely.

Most of the pseudo-interfaces defined by the IDL for the administration of the ORB are defined in the ORBLink module. The associated symbols are thus interned as external symbols in the `orbblink` package.

Except for this all of the mapping is done as specified in the official mapping document.

All of the ORB pseudo-interfaces in excess of what is specified by CORBA itself are defined in IDL itself. Interface definitions in this file are hyperlinked to their semantic definition.

Thus, the Lisp calling sequence for any particular API can be determined directly from the pseudo-IDL.

For example, some of the IDL for message looks like:

```
module ORBLink {...
  pseudo interface Message {
    ...
    enum MessageDirection {incoming,outgoing,unknown};
    readonly attribute MessageDirection direction;
  };
};
```

This is shorthand for the following:

- There is a Lisp class named by the Lisp symbol `orbblink:message`.
- There is a Lisp type named by the Lisp symbol `orbblink:message/messagedirection` which denotes the type comprising the three keywords `:incoming`, `:outgoing` and `:unknown`.
- If `m` is a Lisp symbol bound to an instance of `orbblink:message`, then the invocation:

```
(op:direction m)
```

will return a value of type `orbblink:message/messagedirection`.

Note that this API gives no way actually to create a new message.

It is important remember that the variable `corba:orb` is always bound to an instance of the ORB pseudo-interface.

/*

IDL for ORBLink administrative interfaces

The IDL source in this file, which is in the code font, encapsulates pseudointerfaces to ORB functionality in excess of what is required by the CORBA IDL/Lisp mapping. In order to understand the way in which this file describes Lisp functions, you should be familiar with that mapping. The functionality described in this document, thus, is only necessary for advanced CORBA applications. Each IDL definition in this file is linked to a text description of the semantics of that definition.

*/

```
module ORBLink {
  native value;
  native Condition;

  exception Forward ;
    Object location;
  }
  exception orblink_servant_exception
    any original_condition;
    string message;}

  pseudo interface Junction {
    readonly attribute value socket;
    unsigned long SecondsIdle();
    boolean isOpen();
  };

  pseudo interface ActiveJunction : Junction {
    readonly attribute unsigned long MessagesReceived;
    readonly attribute string RemoteHost;
    readonly attribute unsigned long RemotePort;
    void close();
  };

  pseudo interface PassiveJunction: Junction {};
  pseudo interface ClientJunction : ActiveJunction {};
  pseudo interface ServerJunction : ActiveJunction {};
  typedef sequence ServerJunctionList;
  typedef sequence ClientJunctionList;
  typedef sequence PassiveJunctionList;

  pseudo interface Message{
    enum MessageDirection {incoming, outgoing, unknown};
    readonly attribute MessageDirection direction;
    enum MessageType {Request, Reply, CancelRequest, LocateRequest,
                      LocateReply, CloseConnection, MessageError, Fragment};
    readonly attribute MessageType type;
    readonly attribute junction ForwardingJunction;
  }

  pseudo interface ORB : CORBA::ORB {
```

```

Object _narrow(in Object obj, in Symbol class_name);
void HandleJunctionError (in Junction j, in Condition c);
void HandleJunctionClose (in Junction j);

readonly attribute ServerJunctionList ServerJunctions;
readonly attribute ClientJunctionList ClientJunctions;
readonly attribute PassiveJunctionList PassiveJunctions;
enum ServerJunctionErrorPolicyType {continue, debug, handle};
attribute ServerJunctionErrorPolicyType ServerJunctionErrorPolicy;
attribute boolean HandleJunctionClosePolicy;

enum break_policy_type {return,break};
attribute break_policy_type break_policy;
enum thread_policy_type {singly_threaded, thread_per_request}
enum verbose_level_type {low, high};
attribute verbose_level_type verbose_level;
attribute thread_policy_type thread_policy;
attribute boolean tracing;
attribute unsigned long port;
attribute string domain ;
attribute string host;
readonly attribute string version;
};

};

module CORBA {
  const ORB orb;
  pseudo interface ORB {
    String object_to_string (in Object obj);
    Object string_to_object (in String str);
  };

  pseudo interface Object {
  };

  pseudo interface Proxy (Object) {
    readonly attribute _junction;
  }

  pseudo interface Servant (Object) {
    readonly attribute string _marker;
    void _forward (in Object location) raises (ORBLink::Forward);
  }
}

```

The ORBLink ORB

The global special variable `CORBA:ORB` is always set to the singleton instance of the ORBLink ORB.

The ORBLink ORB is encapsulated in the pseudointerface:

```
pseudo interface ORB : CORBA::ORB {
  Object _narrow(in Object obj, in Symbol class_name);
  void HandleJunctionError (in Junction j, in Condition c);
  void HandleJunctionClose (in Junction j);

  readonly attribute ServerJunctionList ServerJunctions;
  readonly attribute ClientJunctionList ClientJunctions;
  readonly attribute PassiveJunctionList PassiveJunctions;
  enum ServerJunctionErrorPolicyType {continue, debug, handle};
  attribute ServerJunctionErrorPolicyType ServerJunctionErrorPolicy;
  attribute boolean HandleJunctionClosePolicy;

  enum break_policy_type {return,break};
  attribute break_policy_type break_policy;
  enum thread_policy_type {singly_threaded, thread_per_request}
  enum verbose_level_type {low, high};
  attribute verbose_level_type verbose_level;
  attribute thread_policy_type thread_policy;
  attribute boolean tracing;
  attribute unsigned long port;
  attribute string domain ;
  attribute string host;
  readonly attribute string version;
};
```

Most of these attributes and operations are defined in other sections, hyperlinked to their name.

The `version` attribute is bound to a string that denotes the current ORB.

Simply printing the value of `corba:orb` will give version, host, and other useful information.

Stringification

Note that the CORBA pseudo-interface itself defines certain functions on the ORB. By far the most important of these are `object_to_string` and `string_to_object`. See the section on naming for more information on these.

The ORBLink IDL to Lisp Compiler

An IDL file may be compiled by using the `corba:idl` function with a single argument, the pathname of the file to be compiled. By default the preprocessor symbol "LISP" is defined.

Basic usage of `corba:idl`

The effect of the function

```
(corba:idl pathname)
```

is to load into the Lisp image those Lisp values defined by the official mapping that correspond to the IDL file denoted by `pathname`.

This includes:

- constant definitions,
- type definitions,
- class definitions,
- proxy definitions,
- servant definitions,
- struct definitions,
- union definitions, and
- marshalling and unmarshalling functions for each of the data types and operations defined in the IDL file.

Generating fasl files from IDL files: Advanced usage of `corba:idl`

The `corba:idl` function can also be used to generate a `.fasl` file from an IDL file. This `.fasl` can later be loaded into a Lisp image into which ORBLink has been loaded. When the `.fasl` file is loaded, the effect on the Lisp world is the same as if the original IDL file had been passed as sole argument to the `corba:idl` function. The name of the generated fasl file is determined by keyword arguments passed to `corba:idl`. The allowed keyword arguments to `corba:idl` are:

- **`:compile`**
Set to `T` if a `fasl` is to be generated from the argument IDL file. The default is `nil`. If set to `T`, a `fasl` file is generated from the argument IDL file. The default name of this `fasl` file is the same as the concatenation of the name of the IDL file without extension concatenated to the string `"-tmp"` with the file extension replaced by `".fasl"`. Note that by default, when the value of `:compile` is set to `T`, the `fasl` file is generated but is not loaded; thus, it is necessary subsequently to load manually the generated `fasl` file.

- **:compile-and-load**

Set to `T` if a `fasl` file is to be generated from the argument IDL file and subsequently loaded into the running Lisp image. The name of the generated `fasl` file follows the conventions of the `:compile` keyword argument.

- **:lisp-file**

The name of the intermediate Lisp file that is generated. The default is the name (without extension) of the given IDL file concatenated to the string `"-tmp.cl"`. This argument should be set only if either `:compile` or `:compile-and-load` were set to `T`.

- **:retain-lisp-file**

Set to `T` if the intermediate Lisp file is to be retained after compilation; by default it is deleted. This option is included for didactic purposes only; in particular, the operation of recompiling this generated Lisp file by the user is explicitly not supported, as the compilation must be done in a particular environment controlled by the ORB.

- **[all other keyword arguments]**

All other keyword arguments are passed through to `compile-file` without change.

For example, given a file `/a/b/x.idl`, the form `(corba:id1 "/a/b/x.idl" :compile t)` will generate a `fasl` file named `/a/b/x-tmp.fasl`. When loaded into another Lisp world, the effect will be the same as if `(corba:id1 "/a/b/x.idl")` were evaluated.

The Interface Repository in ORBLink

An interface repository is an object implementing the `CORBA::Repository` interface. This is a standard CORBA interface whose IDL is given in the CORBA Core IDL.

The purpose of the interface repository is to maintain type information about IDL files. Once an IDL file is compiled, its definitions can be stored in an interface repository and can be retrieved remotely by other ORBs.

The semantics of the interface repository are specified in the CORBA Core standard as well as in most good books on CORBA. Therefore, we will not discuss the semantics here in detail except to give a few simple examples. Unlike most ORBs, ORBLink naturally and seamlessly can access multiple interface repositories. Each invocation of the IDL compiler creates a new interface repository that can be navigated.

Getting the root interface repository object

The simplest way to obtain the interface repository object is as the value returned by the IDL/Lisp compiler, `corba:idl`.

The script below illustrates navigation of the grid example IDL.

```
; Get the repository object
USER(5): (setq repository (corba:idl "examples/grid/idl/grid.idl"))
#< ORBLINK::CORBA-REPOSITORY-IMPLEMENTATION :DEF_KIND :DK_REPOSITORY @ #x86b680a>

; List all definitions in the repository
USER(6): (op:contents repository :dk_all nil)
(#< ORBLINK::CORBA-MODULEDEF-IMPLEMENTATION :NAME "example" :ID "IDL:example:1.0" :DEF_KIND :DK_MODULE @ #x86b9dca>)

; Get the first (and only) definition
USER(8): (setq moduledef (car (op:contents repository :dk_all nil)))
#< ORBLINK::CORBA-MODULEDEF-IMPLEMENTATION :NAME "example" :ID "IDL:example:1.0" :DEF_KIND :DK_MODULE @ #x86b9dca>

; Get the name of this definition
USER(9): (op:name moduledef)
"example"

; describe the module definition (this returns a struct)
USER(10): (op:describe moduledef)
#< CORBA:MODULEDESCRIPTION :NAME "example" :ID "IDL:example:1.0" :DEFINED_IN "" :VERSION "1.0" @ #x86bd67a>

; List the contents of the module
USER(11): (op:contents moduledef :dk_all nil)
(#< ORBLINK::CORBA-INTERFACEDEF-IMPLEMENTATION :NAME "grid" :ID "IDL:example/grid:1.0" :DEF_KIND :DK_INTERFACE @ #x86bf31a>)

; Get the interface defined in the module
USER(12): (setq interfacedef (car (op:contents moduledef :dk_all nil)))
#< ORBLINK::CORBA-INTERFACEDEF-IMPLEMENTATION :NAME "grid" :ID "IDL:example/grid:1.0" :DEF_KIND :DK_INTERFACE @ #x86bf31a>

; Describe the interface
USER(13): (op:describe interfacedef)
#< CORBA:INTERFACEDESCRIPTION :NAME "grid" :ID "IDL:example/grid:1.0" :DEFINED_IN "IDL:example:1.0" :VERSION "1.0" :BASE_INTERFACES NIL @ #x86c0732>

; Get the repository ID of the interface
USER(14): (op:id interfacedef)
"IDL:example/grid:1.0"

; List the contents of the interface
USER(15): (op:contents interfacedef :dk_all nil)
(#< ORBLINK::CORBA-OPERATIONDEF-IMPLEMENTATION :NAME "_get_width" :ID "IDL:example/grid/_get_width:1.0" :DEF_KIND :DK_OPERATION @ #x86c247a>
#< ORBLINK::CORBA-OPERATIONDEF-IMPLEMENTATION :NAME "_get_height" :ID "IDL:example/grid/_get_height:1.0" :DEF_KIND :DK_OPERATION @ #x86c2602>
#< ORBLINK::CORBA-ATTRIBUTEDEF-IMPLEMENTATION :NAME "height" :ID "IDL:example/grid/height:1.0" :DEF_KIND :DK_ATTRIBUTE @ #x86c37aa>
#< ORBLINK::CORBA-ATTRIBUTEDEF-IMPLEMENTATION :NAME "width" :ID "IDL:example/grid/width:1.0" :DEF_KIND :DK_ATTRIBUTE @ #x86c392a>
#< ORBLINK::CORBA-OPERATIONDEF-IMPLEMENTATION :NAME "set" :ID "IDL:example/grid/set:1.0" :DEF_KIND :DK_OPERATION @ #x86c3aca>
#< ORBLINK::CORBA-OPERATIONDEF-IMPLEMENTATION :NAME "get" :ID "IDL:example/grid/get:1.0" :DEF_KIND :DK_OPERATION @ #x86c3c22>)
```

Getting the interface repository from an object

Given any CORBA object `o`, in theory the invocation:

```
(op:_get_interface o)
```

will return an instance of class `CORBA:InterfaceDef` that can be used (via the `op:containing_repository` method) to get the repository. However, in practice some ORBs either do not implement interface repositories or do not enable them by default. However, this call should always work for any ORBLink object. In fact, it can be used to traverse the repository of all CORBA definitions: given any ORBLink object `oo`, the invocation:

```
(op:containing_repository  
 (op:_get_interface  
  (op:_get_interface oo)))
```

should return a `corba:repository` object that represents all of the CORBA Core definitions.

Implementing operations: smart proxies and synchronization

The main tool for defining implementation methods is the `corba:define-method` macro. The syntax is very similar to that of the usual Lisp `defmethod`, except that the first argument, and only the first argument, may be specialized. The formal syntax of `corba:define method` is described in the mapping document. There are numerous examples in the `examples` subdirectory. A typical `define-method` form might look like the following, taken from `examples/test/test-implementation.cl`.

```
(corba:define-method testexception ((this test-implementation) a)
  (cond
    ((plusp a) a)
    ((zerop a) (/ 1 0))
    (t
     (error 'idltest:exceptionexample :member1 a))))
```

This defines a new implementation of `op:testexception`, which should have been defined as an operation in an IDL interface implemented by the `test-implementation` class. Incoming requests to invoke the operation `testexception` on a `test-implementation` instance will result in evaluation of the `corba:define-method` body.

Signalling exceptions in `corba:define-method`

You can signal a condition within the body of `corba:define-method` form as normal. When the implementation body is invoked remotely, the ORB will catch any signalled exceptions and behave as described by the ORB exception API.

Synchronization

You can interlock method invocations on a particular invocation by using the `:synchronized` keyword after the method name.

Smart proxies

You can use the usual Common Lisp `:before`, `:after`, and `:around` methods. One common usage is to create a smart proxy. For example, consider the IDL:

```
module M {
  interface test{
    long remoteoperation (in long fum);};
};
```

Suppose we want to print a message every time the `remoteoperation` operation on any proxy of class `test` was invoked. We could do this via:

```
(corba:define-method remoteoperation :before ((this M:test-proxy) fum)
  (format t "Proxy: ~s received message remoteoperation with parameter: ~a~%" fum)
  (force-output)) ; always a good idea to call this after formats which may occur in background
```

Now suppose there is a global variable named `*call-remotely*`. We can forward remote requests remotely only when this is T, otherwise we return 0:

```
(corba:define-method remoteoperation :around ((this M:test-proxy) fum)
  (if *call-remotely* (call-next-method)
      0))
```

You can use this, for example, to cache remote state and avoid remote invocations.

Forwarding requests

You can forward the request to another object within the body of a define-method form. The forwarding API is described separately.

Message access

Within the body of a define-method form the `orblink: *message*` special variable is bound to the IIOP Request message that caused the invocation. This functionality is described separately.

Debugging ORBLink processes

The following IDL pertains to debugging APIs:

```
module ORBLink {
  pseudo interface ORB {
    enum verbose_level_type {low, high};
    attribute verbose_level_type verbose_level;
    attribute boolean tracing;
  }
}
```

The `verbose_level` attribute of `corba:orb`, when set to `:high` outputs various information.

The `tracing` attribute of `corba:orb`, when T, prints a message, whose verbosity is governed by the `verbose_level`.

Setting the `verbose_level` to `:high` automatically turns on tracing.

These particular debugging APIs are somewhat crude, of course. For more sophisticated debugging issues, you should can use the usual reconfigurability properties of Lisp.

For example, you can redefine a server method on the fly, which can be used to insert print statements in server methods.

You can also use auxiliary methods to trace execution of particular remote or local invocations.

Exception handling in ORBLink

- Introduction
- Server behavior on implementation exception
- When (op:break_policy corba:orb) is :break
- When (op:break_policy corba:orb) is :return
- Default server behavior
- Default configuration failure modes

Introduction to exception handling in ORBLink

This section describes how to customize the behavior of the ORB when a Lisp implementation of a CORBA interface signals an unexpected exception. This section does not discuss handling of two other kinds of exceptions:

- Customization of I/O exception handling is discussed in the section on connection management.
- Invocation of a CORBA operation on a Lisp client-side proxy may result in a condition being signalled to the client. The behavior of the program in this case can be customized via the standard Common Lisp `handler-case` form and is outside the scope of this document.

The IDL encapsulation for exception handling in ORBLink is encapsulated in the following IDL:

```
module ORBLink {
  pseudo interface ORB : CORBA::ORB {
    enum break_policy_type {return,break};
    attribute break_policy_type break_policy;
  };
};
```

ORBLink server behavior on implementation exception

When a user's Lisp implementation of a CORBA interface signals a condition that is not a subtype of `CORBA:userexception`, or if it returns types that are not consistent with its IDL signature, one of two things occurs:

1. A debugger break loop is entered on the server or
2. A corba system exception is returned to the client.

Which of these occurs depends on the value of the `break_policy` attribute of `corba:orb`:

When (op:break_policy corba:orb) is :break

If the value of the `break_policy` attribute of `corba:orb` is `:break`, then option 1. above will be selected:

```
(setf (op:break_policy corba:orb) :break)
```

A debugger break loop will be started and will offer the user at least the following options:

Debugger break loop options

- Returning a corba SYSTEM EXCEPTION to the client
- Reinvoking the function that signalled the condition.

Thus, the user can fix or modify, and then recompile, the implementation code before re-invoking it.

As such, it is useful important to understand the concepts of dynamic ORBLink server reconfiguration in CORBA server development.

When (op:break_policy corba:orb) is :return

If the value of the `break_policy` attribute of `corba:orb` is `:return`, option 2 is selected. A CORBA system exception will be returned to the invoking client.

```
(setf (op:break_policy corba:orb) :return)
```

Default error handling

The default value of the `break_policy` attribute of `corba:orb` is:

```
:break
```

```
.
```

Default configuration failure modes

The invocation of a debugger by a server thread in response to an undeclared exception signalled by user code will not always result in a useful debugger output.

In particular, if the server is being run from a standard console, for example in a Unix tty shell, the created debugger will attempt to share input with the main Lisp listener which will result in user input being sent to the incorrect thread.

In consequence, when running an ORBLink server from a standard console, you should normally set the value of the `break_policy` attribute of `corba:orb` to `:return`.

In general, we have found the emacs/Lisp environment to be quite powerful for multi-threaded CORBA/ORBLink development.

Naming and persistent IORs

One of the trickiest problems in CORBA is the bootstrap problem: how does a CORBA client get an initial object on which to invoke object references?

Introduction and terminology of naming

Recall that in CORBA, every object has a unique Interoperable Object Reference, or IOR, that can be used to locate the object. The IOR contains information such as the host on which the object resides, the TCP/IP port on which the object is listening for requests, and an object key that distinguishes the object from other objects listening on that port.

An IOR has an opaque string representation, called a stringified IOR that may be read by any CORBA-compliant ORB. An typical stringified IOR (or just IOR for short) looks like:

```
IOR:00585858000000154944c3a69646c746573742f746573743a312e300058585800000010000000000000350001005800000006636f726261009b44000000214f52424c096e6b3a3a636f7262613a333393734383a3a736b656c6574666e202330
```

CORBA itself defines two operations in the ORB pseudo interface for acting on stringified IORs:

```
module CORBA {
  pseudo interface ORB {
    string object_to_string (in Object obj);
    Object string_to_object (in String ior);
  };
}
```

Thus, given a stringified IOR bound to the string `str`, a Lisp proxy for the object represented by the string can be created via:

```
(op:string_to_object corba:orb str)
```

Given an object, its stringified IOR can be formed via:

```
(op:object_to_string corba:orb str)
```

Recall here that `corba:orb` is always bound to the ORB itself.

The simplest way for a client to access the server object on which it is to make invocations is for it to retrieve the stringified IOR of the server object.

ORBLink offers two utility functions to facilitate this.

The function `orblink:read-ior-from-file`, given a filename, returns a proxy for the IOR string stored in that filename.

The function `orblink:write-ior-to-file`, given an object and a pathname, writes the IOR of the given object to the file denoted by its argument.

The definition of these functions are located in the file `examples/ior-io/cl/sample-ior-ior.cl`.

This method for bootstrapping, although simple to understand and test, has several disadvantages, preeminent among which is the need for the client and the server to share access to a file system.

CORBA also defines a naming service. A Name Service is simply a standard CORBA object which contains operations that bind human-readable names to objects.

ORBLink itself will interoperate with other CORBA-compliant name services and, in addition, ORBLink contains a name service that is bundled, with source, with the ORB. The source to the ORBLink name service is located in the directory `examples/naming/`.

However, configuring and using a CORBA compliant name service has two disadvantages:

1. Its API is somewhat complicated (although standardized).
2. The IOR of the name service itself must be published.

The advantage of 1. above is that, although the CORBA Naming API is complicated, there are numerous third-party books that describe it. Once the CORBA Naming API and the IDL/Lisp mapping are learned, the Lisp API to the naming service is immediate.

In particular, if a CORBA naming service is already in use in your organization, this can be a good solution.

Another good solution to the naming problem is for the server to write the IOR to a file as usual, but for the client to retrieve the file via `http`:. This requires that a Web server have access to the file system to which the application server wrote the IOR. Such an architecture is particularly useful when the client is in Java and the server is in Lisp.

Finally, the process of disseminating IORs can be made simpler if the IOR is persistent; that is, if it does not change over even when the process holding the object it represents is started and stopped.

Creating persistent IORs

The IOR of an ORBLink object is formed from three fields:

- The host on which the ORB runs.
- The port on which the ORB is listening
- The marker of the object.

The `host` and the `port` are attributes of the ORB, the object bound to `corba:orb`.

The salient IDL for these attributes is located in the file: `orblink-idl.htm`:

```

module ORBLink{
  pseudo interface ORB : CORBA::ORB {
    ...
    attribute unsigned long port;
    attribute string host;
  };
};

```

These attributes are described in more detail in the section on the ORB pseudo-interface.

Each servant has a readonly attribute, its `_marker`. The name of this attribute is `_marker`, from the IDL

```

module CORBA {
  pseudo interface Servant {
    readonly attribute string _marker; // Only applies to servants
  };
};

```

A CORBA implementation object inherits from `corba:servant`. You can assign a **marker**, a unique name for the implementation object within the ORB, by using the `:_marker` initarg at object creation time. It is an error to assign the same marker to distinct objects.

Thus, the following sample code should return an object whose IOR is constant over different invocations. We assume that the class `grid-implementation` has been defined to inherit from `corba:servant`, for example using

```

(defclass grid-implementation (interface_name-servant)...
  (setf (op:port corba:orb) 50000) ; Set the port on which the ORB listens.
  (setq grid-object (make-instance 'grid-implementation :_marker "GridServer"))
  (corba:object_to_string grid-object) ; get the IOR and start the socket listener

```

The string returned by the call to `object_to_string` should be constant across Lisp world invocations on the same host.

Architecting naming services

If you do not already have a Naming Service configuration, we recommend the following architecture.

1. Set up a single factory object for your application. This server will simply provide the IOR of the actual entities in your application. You can customize and design it as you wish, but it is better to separate it from the application logic. For example sample IDL might be:

```

module ObjectFactory{
  interface ApplicationFactory{
    Object GetApplicationInstance();};};

```

2. Implement the `GetApplicationInstance` operation in the IDL. Start the `ApplicationFactory` server and publish its IOR. There are several ways to publish the IOR as described above:

- Write it to a file that can be read by all the clients.
- Store it in a Web Server.
- Start the `ApplicationFactory` server as a persistent IOR and hardcode the IOR into application code or into the applet parameter fields in HTML.

The IDL/Lisp Standard and ORBLink

The CORBA IDL/LISP proposed standard mapping allows the implementor flexibility in the implementation of various types and functions. The purpose of this document is to describe the choices we have made.

Sequence handling

This ORB always unmarshals values corresponding to the IDL sequence type as vectors. However, a list may be used anywhere a sequence is expected.

float and double types

`corba:float`, corresponding to the IDL `float` type, denotes the type `single-float`.

`corba:double`, corresponding to the IDL `double` type, denotes the type `double-float`.

Operation mapping

ORBLink implements CORBA operations as generic functions. An operation named "foo" will be mapped to a method on a standard generic function named `OP:FOO`.

The signature of the generic function is `(T &REST ARGS)`.

User code may rely on the fact that `OP:FOO` is a funcallable object and may freely use auxiliary methods with such objects. However, Franz may change the low-level implementation of such functions in the future and user code should not rely on the metaclass or the signature of such generic functions.

Unimplemented features

The following aspects of the CORBA Core are unimplemented:

1. DII/DSI
2. Wide strings, wide characters
3. Extended numeric types (long double, long long)
4. Fixed types
5. POA
6. Contexts

Note on on enum types

According to the CORBA 2.2 specification, an enum type does not begin a new naming scope. This means that enum members themselves define a new type in the namespace of the IDL construct containing the enum type definition. The CORBA 2.3 specification may change this counter-intuitive rule. In any case, we follow the CORBA 2.2 specification so that users must make sure that enum members do not conflict with names in a containing scope.

Connection Management in ORBLink

Connection management introduction and terminology

Connection management refers to the set of APIs that perform administrative functions that handle socket opening and closing. Using the connection management API, an application can determine the set of open sockets, can close them when applicable, and can set hooks that are called when a particular connection is closed.

All of the connection management APIs are encapsulated in IDL.

Junction

The ORB interfaces to TCP/IP sockets through the abstract interface `ORBLink::Junction`:

```
module ORBLink{
  pseudo interface Junction {
    readonly attribute value socket;
    unsigned long SecondsIdle();
    boolean isOpen();
  };
};
```

The attributes and operations supported by the `ORBLink::Junction` pseudointerface (which is to say, the Lisp class named `orblink:junction`):

- The `socket` attribute of `ORBLink::Junction` is an opaque Lisp type corresponding to the underlying socket stream.
- The `SecondsIdle()` operation gives the number of seconds the junction has been idle. It is reset to 0 on creation or I/O activity, but information requests do not result in resetting the idle time.
- The `isOpen()` operation returns T if and only if the junction is actively forwarding messages, that is if its state is open. This state can be set to closed by the `ActiveJunction::close()` operation.

Subclasses of Junction

```
module ORBLink {...
  pseudo interface ActiveJunction : Junction {
    readonly attribute unsigned long MessagesReceived;
    readonly attribute string RemoteHost;
    readonly attribute unsigned long RemotePort;
    void close();
  };
  pseudo interface PassiveJunction : Junction { };
  pseudo interface ClientJunction : ActiveJunction { };
  pseudo interface ServerJunction : ActiveJunction { };
};
};
```

Any junction instance is an instance of one of three disjoint classes:

1. `Orblink:ClientJunction`

An instance of `ORBLink:ClientJunction` is responsible for forwarding messages, normally request messages, from a CORBA client to a CORBA server.

2. `ORBLink:ServerJunction`

An instance of `ORBLink:ServerJunction` is responsible for forwarding messages, normally replies, from a CORBA server to a CORBA client.

3. `ORBLink:PassiveJunction`.

An instance of `ORBLink:PassiveJunction` is responsible for listening to connection requests from prospective clients and allocating Server junctions as necessary to handle the resulting connections.

ActiveJunction

An instance of `Orblink:ActiveJunction` is either an instance of `orblink:clientjunction` or an instance of `orblink:serverjunction`. The attributes supported by the pseudo-interface `ORBLink::ActiveJunction` are:

- The `MessagesReceived` attribute gives the number of messages that have been received.
- The `RemoteHost` attribute gives the name of the host to which the active junction is connected.
- The `RemotePort` attribute is the remote port of the corresponding socket.

An instance of `orblink:activejunction` also supports the `close` operation. When invoked on a junction, the `close` operation:

- closes the junction - future invocations of the `isOpen()` operation will return `nil`
- Terminates any associated threads, typically threads listening for input, and
- closes any associated streams, namely the value of the `socket` attribute if it is an open stream.

An active junction also closes itself if its associated `socket` stream signals an I/O error, including an EOF error.

- When a `ServerJunction` is closed, it removes itself from the list of `ServerJunctions` maintained by the ORB in its `ServerJunctions` attribute; it cannot be reused.
- When a `ClientJunction` junction is closed, however, it is reopened when a new `Request` is invoked through that junction.

Determining the available junctions

The ORB itself offers facilities for listing the available junctions and for customizing the behavior of a junction on closure:

```

module ORBLink {
  pseudo interface ORB : CORBA::ORB{
    readonly attribute ServerJunctionList ServerJunctions;
    readonly attribute ClientJunctionList ClientJunctions;
    readonly attribute PassiveJunctionList PassiveJunctions;
  };
};

```

The `ORBLink::ORB::ServerJunctions`, `ORBLink::ORB::ClientJunctions`, and `ORBLink::ORB::PassiveJunctions` attributes contain lists of the operational junctions of the appropriate type.

Thus, the forms:

```

(op:serverjunctions corba:orb)
(op:clientjunctions corba:orb)
(op:passivejunctions corba:orb)

```

Will return lists (or sequences) of the server, client, and passive junctions.

Junction close policies

The behavior of junctions on closure is determined by the following pseudo-IDL:

```

module ORBLink {
  pseudo interface ORB : CORBA::ORB{
    attribute boolean HandleJunctionClosePolicy;
    void HandleJunctionClose (in Junction j);
  }
}

```

When the value of the `HandleJunctionClosePolicy` attribute of the `corba:orb` singleton is `nil`, junction closure operates normally.

When the value of the `HandleJunctionClosePolicy` attribute of the `corba:orb` singleton is `T`, on the other hand, after a junction `j` is closed the `HandleJunctionClose` operation of the `CORBA:ORB` object is invoked with parameter of `j`. In this case, the user should override the definition of `HandleJunctionClose`.

For example, the following set of definitions will print a message whenever a junction is closed:

```

(corba:define-method HandleJunctionClose ((orb ORBLink:ORB) junction)
  (format t "HandleJunctionClose: closed junction: ~s ~%" junction)
  (force-output)
)

(setf (op:HandleJunctionClosePolicy CORBA:ORB) T)

```

Junction error policies

The behavior described in this section is experimental and has not been tested.

The behavior of the ORB on a server junction error (that is, an I/O error, as contrasted with an error signalled by a servant) is also customizable. It is encapsulated in the ORB IDL:

```
module ORBLink {
  pseudo interface ORB : CORBA::ORB{
    enum ServerJunctionErrorPolicyType {continue, debug, handle};
    attribute ServerJunctionErrorPolicyType ServerJunctionErrorPolicy;
  };
};
```

Because a junction error results in a junction close, normally customization of the close method is sufficient.

The server junction error handling is determined by the `ServerJunctionErrorPolicy` attribute of `CORBA:ORB`:

1. If this value is `:continue`, the error is ignored.
2. If the value is `:debug`, a debugger is invoked.
3. If the value is `:handle`, the `HandleJunctionError` operation on the `CORBA:ORB` object is invoked with parameters the server junction that caused the error and the error itself.

The default value of the `ServerJunctionErrorPolicy` is `:continue`.

Client junction errors are normally signalled back to the invoking client; thus, client junction error customization is not exposed in this API.

The Message pseudo-interface

The IDL for the message interface in ORBLink is:

```
pseudo interface Message{
  enum MessageDirection {incoming,outgoing,unknown};
  readonly attribute MessageDirection direction;
  enum MessageType {Request,Reply,CancelRequest,LocateRequest,
                    LocateReply,CloseConnection,MessageError,Fragment};
  readonly attribute MessageType type;
  readonly attribute junction ForwardingJunction;
}
```

The corresponding Lisp class is named `ORBLink:message` and instances of that class represent IIOP messages.

Suppose `m` is bound to an instance of `ORBLink:message`. Then the form

```
(op:direction m)
```

corresponding to the `direction` attribute will return a value of type `ORBLink:MessageDirection`, that is, one of `:incoming`, `:outgoing`, or `:unknown` depending on whether `m` represents an incoming, outgoing, or message of unknown direction.

The form

```
(op:type m)
```

corresponding to the `type` attribute of `m` returns a keyword of type `ORBLink:MessageType`, a member of `(:request :reply :cancelrequest :locaterequest :locatereply :closeconnection :messageerror :fragment)`.

The `ForwardingJunction` attribute of `m` holds an instance of class `ORBLink:Junction` which represents the junction from which the message was received (if an incoming message) or to which the message is being sent (if an outgoing message).

Only Request messages can actually be obtained using exposed APIs.

A Request message (a message whose `type` attribute is `:request`) is obtained as follows.

When a `ServerJunction` receives an IIOP Request message from a client, the operands are dispatched to the appropriate local implementation object which then executes the body of the `corba:define-method` corresponding to the operation requested by the message. Within the dynamic scope of that body, the special variable `orbLink:*message*` is bound to the corresponding request message.

The appropriate server junction can then be obtained from the value of the `ForwardingJunction` attribute of `message`.

The IDL and implementation in `examples/test/test.idl` and `examples/test/test-implementation.cl` include a simple example for accessing the message from the body of a `corba:define-method` form.

The relevant IDL is in `examples/test/test.idl`:

```
module idltest{ interface test
    oneway void testmessage (in unsigned short delay);
```

The associated implementation class in `examples/test/test-implementation.cl` is:

```
(defclass test-implementation (idltest:test-servant)
  ((message :accessor get-message)))
```

The implementation of the operation defined in the IDL is given by:

```
(corba:define-method testmessage ((this test-implementation) delay)
  (sleep delay)
  (format t "testmessage: got message of: ~s~%" orblink:*message* this)
  (force-output)
  (setf (get-message this) orblink:*message*)
  )
```

(The `delay` can be used easily to verify that `orblink:*message*` is bound to different messages in different threads).

Thus, the `serverjunction` corresponding to a message can be obtained via:

```
(op:forwardingjunction orblink:*message*)
```

In conjunction with the `handlejunctionclose` operation and the `HandleJunctionClosePolicy` attribute in pseudo interface `ORBLink::ORB`, these features allow the user, for example, to determine which implementations correspond to which server junctions and to perform local cleanup if desired.

Threading in ORBLink

The IDL associated with threading is in the `ORBLink:ORB` pseudoInterface.

```
module ORBLink{
  pseudo interface ORB : CORBA::ORB {
    enum thread_policy_type {singly_threaded, thread_per_request};
    attribute thread_policy_type thread_policy;
  };
};
```

The enum statement encodes the fact that there is a type named `ORBLink:thread_policy_type` comprising the keywords `:singly_threaded` and `:thread_per_request`.

Thus:

```
(typep :singly_threaded 'orblink:thread_policy_type)
--> T
```

The ORB has an attribute named `thread_policy` whose value is always a member of that type.

The value of the attribute can be retrieved as follows:

```
(op:thread_policy corba:orb)
--> :THREAD_PER_REQUEST
```

The value of the attribute can be set using standard `setf` syntax:

```
(setf (op:thread_policy corba:orb) :thread_per_request)
```

Thus, the IDL definitions describe how to access certain values but they do not describe the meaning of the values themselves; that is the purpose of this document.

When the ORB handles an incoming request, it spawns a separate Lisp thread, an instance of `mp:process`, if the value of the `thread_policy` attribute of `corba:orb` is `:thread_per_request`.

This is the default and is the recommended settings. If the value of the attribute is set to `:singly_threaded`, however, then the request is executed in the same thread as was used to read the request from the wire.

Forwarding requests to another object

You can use the `_forward` operation in the `Object` pseudo interface to forward a request to a different object:

```
pseudo interface servant (Object) {  
    void _forward (in Object location) raises (ORBLink::Forward);  
}
```

Within the body of a `corba:define-method` definition corresponding to an implementation object `r`, the function invocation

```
(op:_forward r p)
```

will forward to the object designated by `p` the request that was received by the object `r`.

This functionality only works if the original request was received remotely by `r`.

It is implemented at the IIOP level by returning to the invoker a reply of type `LOCATION_FORWARD` with the IOR of `p` in the IIOP message body. All subsequent requests on that proxy (which can be in the address space of a non-Lisp ORB) which forwarded the original request to `r` will be routed directly to `p`.

`op:_forward` signals an `ORBLink:Forward` condition which is handled by the ORB when servicing a remote request. This implementation detail is normally transparent to the user and should not be relied upon.

Dynamic reconfiguration of ORBLink servers

An ORBLink server may be dynamically reconfigured in a number of ways.

Redefinition of implementation methods

A `corba:define-method` form may be reevaluated at any time. The most recent definition will be used.

One common such usage occurs when an implementation has signalled an exception for with the ORB is configured to enter a debug loop. From the debug loop, the implementation may be re-invoked; if the implementation has been modified, the most recent implementation definition will be used.

Recompilation of IDL files

An IDL file may be edited and recompiled using `corba:idl`. The semantics are analogous to the semantics of normal Lisp redefinition of types.

This is most useful when a server interface has been modified to accept *additional* operations. Existing clients will continue to work unchanged and new clients can take advantage of the new operations.

However, if the signature of an existing operation was altered, care must be taken to avoid clients using the old operation definition.

The tutorial works though an example of dynamically adding an attribute to an existing server.

Reconfiguration of clients

Of course, clients can also be reconfigured simply by recompiling the IDL. Care must be taken to ensure that the client is not using an old operation with an incompatible signature. That is, in order to modify an operation to an incompatible signature, the IDL in both the server and the client must be recompiled.

Other means for server configuration

Sometimes handlers for certain server events, like errors and junction closure can be defined.

The ORB itself offers a few additional run-time configuration options.

Using ORBLink to bridge Lisp and Java

A comprehensive discussion of the issues involved in bridging Java and Lisp using ORBLink requires in a certain sense full discussion of the installation and use of Java ORBs.

Fortunately, there are several good books on CORBA and Java available, for example, **Programming with Visibroker**, by Doug Pedrick, Jonathan Weedon, Jon Goldberg, and Erik Bliefield, Wiley, 1998.

A tutorial walking through a Lisp-Java application is available [here](#).

Java applets and CORBA

You can use CORBA to connect a Java applet to Lisp, but doing so is more complicated than hooking up a Java application to a Lisp program. The reason it is more complicated is that a Java applet imposes security restrictions on network connections. The nature of these security restrictions tends to be highly enterprise-dependent, depending on such variables as:

1. The exact version of the browser your users are using, and what Java ORB you want to use.
2. The security configuration of the browser.
3. The version of JDK you are using; whether you are using the Java activator
4. The firewall configuration at your site.
5. Whether internet or intranet access is desired.
6. Should the applet be usable to remote users who will tunnel through their own firewall?

There is no "one perfect solution" for all applications at this time. In general, for example, the you can use http tunnelling (using, for example, the Visigenic tunnelling tools) to allow internet applet users to tunnel through their own firewalls, but this often imposes an unacceptable performance overhead for things like callbacks. Browsers tend to support different versions of the JDK as well; of course, if you are willing to use the Java plug-in, things can be somewhat simpler.

However, if you have control over the browser configuration and are running within an intranet so that firewall configuration is not a problem, hooking up to an applet is much easier. Still, we suggest you begin by getting your application working correction as a Java application and only later addressing the more subtle security and browser issues that are required to get an applet solution to work reliably.

Bridging Allegro CL Applications to Java with ORBLink

This example illustrates how to connect a simple Java GUI to Lisp server using CORBA. We will use ORBLink as the Lisp ORB and Visibroker for Java as the Java ORB.

In order to run the example, you'll need Java, which you can get from JavaSoft, or elsewhere. The example we give today uses the JavaSoft version of the Java VM version 1.1. To use Java CORBA requires a third party ORB, and we use Visibroker. (Of course, CORBA support will be bundled into the JDK 1.2, although the Visibroker ORB is nonetheless more industrial strength than JavaSoft's bundled ORB). You can get trial copies of Visibroker for free from Inprise.

The example application is borrowed from February/March Java Pro article written by Luke Andrew Cassidy-Dorion (the article and the Java code are used with permission). All code can be found here. There is a Makefile for making the Java side, but we'll actually go through the steps here.

This example is a distributed Chat application. The Chat application consists of an arbitrary number of Java based Listeners, each of which permits you to type in comments, and an Allegro CL server, which broadcasts comments to all the other Listeners.

Designing the Application

CORBA IDL (Interface Definition Language) is a specification language that describes behavior, not implementation. We give as an example a simple Chat Room server and Chat Room listeners application. ChatListenerI is implemented by the Java code and ChatServerI is implemented in Lisp. When the server calls messageReceived the actual code runs remote in the Java client. ChatServerI is implemented in the Lisp server. When a new Java client listener starts up, it calls addListener to register itself with the server. When input gets typed into a Java client listener, the client calls sendMessage to send its input to the server which, in its turn, calls messageReceived (with the text it just got from the listener that called sendMessage) on each of the registered listeners. The code for messageReceived in each listener displays the text of the message.

The interface definitions are below

```
module chat{
  interface ChatListenerI{
    void messageReceived (in string message);
  };
  interface ChatServerI{
    void addListener (in ChatListenerI listener);
    void sendMessage(in string message);
  };
};
```

The Server Implementation

Here is the server implementation (it's all in the file *ChatServer.cl* which is in the files you have downloaded). We define a class my-server, which will inherit from a class automatically generated by the Allegro OrbLink IDL compiler. By convention, this generated class chat:ChatServerI-servant is in a package with the same name as the IDL module and is named for the interface with -servant attached.

```
(defclass my-server (chat:ChatServerI-servant)
  ((listeners :initform nil :accessor get-listeners)))
```

The class `my-server` has a slot, `listeners` that will contain all the registered listeners. To complete the implementation, we need to define the methods `addListener` and `sendMessage`. Adding a listener just pushes the new listener onto the slot (remember, when a Java listener instantiates itself, the first thing it does is call `addListener`). The method `sendMessage` takes the message sent by a listener and broadcasts it to all the listeners by successively calling `messageReceived` on each listener in turn. The code for displaying the message in each listener is the implementation code for `messageReceived`, which, of course, is written in Java.

```
(corba:define-method addListener ((this my-server) listener)
  (push listener (get-listeners this)))
(corba:define-method sendMessage ((this my-server) message)
  (dolist (listener (get-listeners this))
    (op:messageReceived listener message)))
```

The Client Implementation

On the Java side, things are a bit more complex and we won't show the full implementation, just a portion of the initialization method that shows how each listener finds and registers itself with the server.

```
private void doConnect(){
    orb = ORB.init();
    boa = orb.BOA_init();
    org.omg.CORBA.Object obj = IorIo.resolve(orb, filename); // filename is the location of the IOR
    server = ChatServerIHelper.narrow(obj);
    ChatListenerI listener = new Listener();
    boa.obj_is_ready(listener);
    server.addListener(listener);
    // ... more stuff ... //
}
```

The way the Java code finds the proxy for the server is to use an *IOR* or *Interoperable Object Reference*, which has the information about where the server is and how to connect to it. An *IOR* is an encoded string of digits containing things like the IP address, the port number, and any other information needed to connect to a server. In this example, as you will see a little later, each Java client is launched with the filename containing the *IOR* as an argument. The Lisp server is responsible for writing the *IOR* into the file when it starts up.

A diagram of the architecture illustrates how it will all work:

Chat Architecture Graphic

Starting up the Lisp Chat server

So here's how we would start up the server (assuming you have changed into the directory where you have the code) and started up an Allegro CL with Allegro OrbLink loaded in:

```
(corba:idl "chat.idl")      ;; load in the IDL and define the interfaces
(load "ChatServer.cl")    ;; load in the chat-server implementation

;; start the server
(setq server (make-instance 'my-server))
;; and write out the IOR
(orbink:write-ior-to-file server "chat.ior")
```

Starting up the Java Chat clients

In the directory where you have put the sample code, there is a Makefile which will automatically compile the Java code, but here are the steps spelled out.

1. Generate the Java code from chat.idl by using the command:
idl2java chat.idl
2. Generate the application using the Visibroker wrapping for the java CORBA compiler:
vbjc ChatClient.java IorIo.java

You are now ready to launch clients using: *vbj ChatClient chat.ior*

Your CORBA based client/server chat application is up and running!

Reporting bugs in ORBLink

To report a bug, set the `verbose_level` attribute of `corba:orb` to `:high` as follows:

```
(setf (op:verbose_level corba:orb) :high)
```

and send a script created via `dribble-bug` to

`orbblink@franz.com`

To use `dribble-bug`, just invoke

```
(dribble-bug filename)
  [code to execute]
(dribble)
```

The script will be created in the file `filename`.

If other ORBs are involved, please include the appropriate source code and IDL code for them as well.

Glossary

CORBA has a lingo all its own. Here are some of the commonly used terms we mention in this document. These definitions are not complete and contain inaccuracies for the sake of simplicity. For more detailed information, see a text on CORBA.

Attribute

Technically an attribute is a shorthand for two operations, a setter and a getter, unless it is readonly, in which case it is shorthand only for the getter.

Client

A process that makes requests on a CORBA object.

exception

IDL data type similar to exceptions in C++ or Java and conditions in Lisp.

Forwarding

Instead of responding to a request a CORBA object may designate another object to which this request, and all future requests, are to be forwarded.

Pseudo-interface

A pseudo-interface is specified in IDL but does not obey all the mapping rules. Typically a pseudo-objects (instance of pseudo-interfaces) must be local to the process. Thus, a pseudo-interface has no corresponding stubs or servant classes generated.

IDL

Interface Definition Language. Simple language for specifying data types that can be exchanged by CORBA compliant programs. IDL files by default have the extension .idl.

IDL Compilation

Conversion of an IDL file into a set of Lisp definitions. Normally refers by default to the loading of these definitions into the current Lisp world.

IIOP

Internet Interoperability Object Protocol. The TCP/IP protocol by which IDL data types are exchanged.

Implementation

In Lisp, a Lisp class that can respond meaningfully to the methods corresponding to the operations defined in the interface.

Interface

An interface is an IDL construct that denotes a set of operation signatures to which an object that implements that interface must respond. Similar to abstract class.

Interface Repository

CORBA Object that holds type definitions.

IOR

IOR stands for Interoperable Object Reference. It is a data structure associated with a CORBA object that contains enough information to locate that object from anywhere on the network. IOR is often used informally to mean *stringified IOR*.

Mapping

The correspondence between IDL and the native data types in some language.

Marshalling

Conversion of a CORBA value into a stream of bytes, typically to be sent over TCP/IP.

Message

IIOP communicates via IIOP messages. The most usual are Request and Reply.

module

top-level scoping construct in IDL, corresponds to packages in Lisp. Modules may be nested.

Naming

The association of a human readable or logical name with an IOR.

Object

A reference to an implementation of an interface

OMG

Object Management Group. Consortium of industry, government and academic entities responsible for standardization of CORBA.

Operation

The IDL notion of a method is called an operation.

ORB

Object Request Broker. Program component that handles communication from a process to other processes using IIOP or other CORBA protocols. An ORB is usually used to refer to the complex of library functionality that supports CORBA for a language, such as debugging tools and IDL compilation.

ORBLink

ORB for Allegro Common Lisp

Persistent IOR

An IOR which is valid beyond the lifetime of any particular server process (i.e, Lisp World)

Proxy

A placeholder for a remote object implementation. When requests are received by the proxy they are forwarded to the remote object that it represents via IIOP.

Servant

In Lisp, the instance of a class that implements an interface.

Skeleton

Same as Servant

Server

Process that maintains one or more servant objects that are invoked by a client. Sometimes the servant object itself is called a server.

Socket

Low-level programming interface to TCP/IP. Sockets are abstracted away by the junction interface in ORBLink.

Stringified IOR

A standardized string representation of an IOR. Any stringified IOR may be interpreted by any CORBA ORB in any language, on any machine.

TCP/IP

Low-level protocol used to communicate between machines; normally supported directly by the operating system.

Smart proxy

A proxy that only forwards some requests over IIOP and computes others using local information.

struct

IDL data type similar to C struct

thread

Lightweight autonomous concurrent execution context. In Lisp these are called "processes" which is distinct from the Unix meaning of the term.

union

IDL data type similar to C union

Unmarshalling

Conversion of a stream of bytes into the CORBA value that it represents.

Introduction

Table of Contents

Introduction	1
CORBA-Enabling Allegro CL Applications	2
Installation	6
Mapping IDL to Common Lisp	7
Summary of the IDL/Common Lisp Mapping	8
Lexical conventions	13
Getting started	14
Using the IDL Administrative interfaces	21
IDL for ORBLink administrative interfaces	22
The ORBLink ORB	24
The ORBLink IDL to Lisp Compiler	25
Interface Repository	27
Implementing operations	29
Debugging ORBLink processes	31
Exception handling in ORBLink	32
Naming and persistent IORs	34
The IDL/Lisp Standard and ORBLink	38
Connection Management	40
Message pseudo-interface	44
Threading	46
Forwarding	47
Dynamic reconfiguration	48
Java	49
Java - an example	50
Reporting bugs	53
Glossary	54