NAME

Combat − Tcl CORBA Object Request Broker

SYNOPSIS

package require Tcl 8.1
package require Itcl 3.0
package require combat ?0.7?

corba::init ?*options ...*?
corba::resolve_initial_references *name*
corba::list_initial_services
corba::register_initial_reference *name obj*
corba::string_to_object *string*
corba::object_to_string *handle*
corba::release ?*typecode*? *value*
corba::duplicate ?*typecode*? *value*
corba::dii ?*options*? *handle spec* ?*args*?
corba::request *cmd* ?*args*?
corba::const *id*
corba::type *cmd* ?*args*?
corba::throw *exception*
corba::try *m-block* ?**catch {***exception* ?*var*?**}** *e-block*? ... ?**finally** *f-block*?
combat::ir *cmd args*

DESCRIPTION

The **Combat** package provides a CORBA Object Request Broker (ORB). Using **Combat**, you can communicate with any CORBA server using the standard IIOP protocol. This is not only useful for test-driving your servers, but also for writing CORBA-enabled applications, especially mixed with a Tk GUI.

Object references are represented by *handles*, which are Tcl commands that front for invocations. This looks quite natural:

```
set bank [corba::string_to_object IOR:...]
set account [$bank create]
$account deposit 1234
puts "Balance is [$account balance]"
```

In this example, **$bank** and **$account** are *handles* (strictly speaking, these are Tcl variables that contain a command name). In an invocation, the first parameter is the operation name, and following parameters are arguments to that operation. The format of arguments is described in the section *Language Mapping* below.

Combat works by reading information about the object's interface from the Interface Repository. IDL files must be loaded into an Interface Repository before they can be used. Read the *Interface Repository* section below for more information.

COMMANDS

corba::init ?*options ...*?

Initializes the ORB. The command accepts an arbitrary number of options. Options starting with the *-ORB* prefix are processed (see below), all list containing all other options is then returned. **corba::init** can therefore be called, at the beginning of a script, as

set argv [eval corba::init $argv]

to remove all ORB-specific command-line parameters from the command line. The call to **corba::init** is optional. All other commands implicitly call **corba::init** with an empty parameter list if the ORB was not initialized before.

The following options are accepted:

-ORBInitRef *name=value*

Sets the initial reference *name* (see **corba::resolve_initial_references**) to *value*. The *value* can have any format that is acceptable to *corba::string_to_object*.

-ORBDefaultInitRef *value*

Sets the default initial reference. See **corba::resolve_initial_references** for more information.

-ORBDebug *level*

Enables debugging output (which is sent to stderr). Valid values for *level* are *giop* (GIOP message exchange), *iiop* (IIOP connection handling), *transport* (raw GIOP data), and *all* (for all of them).

-ORBServerPort *port*

Initializes a TCP socket to listen on port *port* for incoming connections. This option can be repeated multiple times to listen on several ports. If this option is not present, a port will be selected automatically if the **RootPOA** is accessed for the first time.

-ORBGIOPMaxSize *value*

Limits the maximum acceptable size of incoming GIOP messages. By default, GIOP messages are accepted regardless of their size. This allows a denial of service attack on a server by sending a huge message, eventually causing memory exhaustion. If set, GIOP messages whose size (in octets) exceeds *value* cause the connection to be dropped. Each connection will also accept up to *value* octets of GIOP fragments. The size of outgoing messages is not limited by this option.

-ORBNativeCodeSet *value*

Sets the native codeset to be used and advertised as SNCS-C. Usually, the native codeset is determined from **encoding system**. **value** can be an OSF registry value or Tcl encoding name.

Any other options that start with -*ORB* cause an error.

corba::resolve_initial_references *name*

Returns the initial reference *name* as set upon initialization, as a handle (i.e. the value from -*ORBInitRef* is passed to **corba::string_to_object**). If no such initial reference is configured, the default initial reference, if configured, and *name* are concatenated and passed to **corba::string_to_object** to return a handle. If this fails, too, the *CORBA::ORB::InvalidName* exception is thrown.

corba::list_initial_services

Returns a list of names configured as initial references.

corba::register_initial_reference *name obj*

Registers the handle *obj* as an initial reference. *name* will appear in the result of **list_initial_services** and can be used with **resolve_initial_references** to retrieve the handle.

corba::string_to_object *string*

Converts the *string* to an object reference, and returns a handle for that object reference. Several formats for *string* are accepted:

IOR: classic CORBA format for stringified object references, a very long string consisting of the IOR: prefix and data in hex.

corbaloc::*host***:***port*/*ObjectKey*

Format containing the IP address, port number and object key of the target object.

corbaname::*host***[:***port***][/***Key***][#***Name***]**

Format referencing an entry in the Naming Service by IP address. If *port* is omitted, 2089 is used. *Key* is the object key of the Naming Service. If omitted, **NameService** is used. *Name* identifies an entry in the Naming Service. This entry is read and returned. If omitted, the reference of the Naming Service itself is returned.

file://[*host***]***path*

Reads the given file. *host* should be empty or **localhost**. *path* is an absolute path name. The contents of this file are read and then passed to **corba::string_to_object**.

http://...

The referenced URL is pulled (for this, the **http** package must be present). Its contents are then passed to **corba::string_to_object**.

corba::object_to_string *handle*

Converts the object reference represented by *handle* into a string, using the **IOR:** *format.*

corba::release ?*typecode*? *value*

Releases a handle. Handles (as returned from e.g. **corba::string_to_object**) must be released when no longer used, or memory leakage occurs. If *typecode* is omitted, *value* must be a handle, which is then released. If *typecode* is present, *value* must match that typecode. In that case, all handles contained in *value* are released.

corba::duplicate ?*typecode*? *value*

Duplicates a handle. This method is primarily for use in server applications, see the note in the *Server Side Scripting* section. If *typecode* is omitted, *value* must be a handle, which is then duplicated; the new handle, which points to the same object reference, is returned. If *typecode* is present, *value* must match that typecode. In that case, all handles contained in *value* are duplicated, and a duplicate of the complete *value* is returned.

corba::dii ?*options*? *handle spec* ?*args*?

Initiates a remote invocation using the *Dynamic Invocation Interface*. Invocations as described in the **Invocations** section normally pull type information from the Interface Repository, as described in the *Interface Repository* section below. In contrast, an invocation using **dii** does not require type information in the Interface Repository; here, type information is passed along with each invocation in the *spec* parameter, which is a list composed of three or four elements. The first element is the typecode of the return value. The second element is the name of the operation to be invoked. The third element describes the parameters. The fourth element is a list of exception typecodes that this operation may throw. The parameter description is a list that contains one element per parameter. Each parameter is described by a list of two elements. The first element is either **in**, **out** or **inout**, and the second element is the typecode of the parameter type.

options can be either **-async** or **-callback** to start an asynchronous request; see below for more information.

corba::request *cmd* ?*args*?

Manages asynchronous requests. See the **Invocations** section below for information about initiating asynchronous requests using the **-async** and **-callback** options to an invocation. *cmd* and *args* can be one of

get *aid* This operation enters the event loop until the asynchronous request identified by *aid* is finished, and then returns its result. All variables corresponding to **in** and **out** parameters on the initial invocation are set in the current context.

poll *aids ...*

Checks whether any of the asynchronous requests identified by *aids* has finished yet. If yes, its identifier is returned, and a subsequent **get** for that identifier is guaranteed not to block. If none of the given asynchronous requests has finished, an empty string is returned.

wait [*aids ...*]

Enters the event loop until any of the asynchronous requests identified by *aids* finishes. If no parameter is given, all outstanding asynchronous invocations qualify. The identifier for one finished asynchronous invocation is returned.

corba::const *id*

Looks up the *id* in the Interface Repository. *id* must be the Repository Id or absolute name of a constant definition. The value of that constant is then returned as an **any** value.

corba::type *cmd* ?*args*?

Handles type definitions. *cmd* and *args* can be one of

of *id* Looks up the *id* in the Interface Repository. *id* must be the Repository Id or absolute name of a type definition. The typecode of that type is returned.

match *typecode value*

Checks whether *value* matches the *typecode*, and returns the result as either 1 (matches) or 0 (does not match).

equivalent *tc1 tc2*

Checks whether the two typecodes *tc1* and *tc2* are equivalent, and returns the result as either 1 (equivalent) or 0 (not equivalent).

corba::throw *exception*

Throws the CORBA exception *exception*. This is a convenience operator; since exceptions use Tcl's normal error handling, you could also throw exceptions using **error**. See the *Language Mapping* section for information on the format of exeptions.

corba::try *m-block* ?**catch {***exception* ?*var*?**}** *e-block*? ... ?**finally** *f-block*?

Provides a more convenient means of catching CORBA exceptions than Tcl's native **catch** command (all CORBA exceptions can be caught with **catch**, if preferred). First, the *m-block* is executed. If an exception occurs, a matching **catch** clause, whose *exception* value matches the exception's Repository Id, is looked for. If found, its *e-block* is executed, initializing the variable denoted by *var*, if present, to the exception value. Whether or not an exception has occured, and whether or not this exception has been handled by a **catch** clause, the optional *f-block* is then executed.

If an error occured in the *f-block*, that error is returned. If an exception has occured, and it has been handled by a **catch** clause, the result from executing its *e-block* are returned. Otherwise, the result from executing *m-block* is returned.

To catch all exceptions, *exception* can be set to **...** (three dots, similar to Java or C++).

Because exceptions are a specialization of Tcl errors, **corba::try** also responds to Tcl errors. If a Tcl error occurs in the execution of *m-block*, the **finally** clause will also be executed. Tcl errors are caught in a **catch** clause if **...** is used as *exception*.

If there are no **catch** clauses, an implicit clause that catches **...** is used.

combat::ir *cmd args*

This command can be used to load interface information into a local Interface Repository. At the moment, the only supported value for *cmd* is *add*. *args* must be a string which is the result of processing an IDL file with the *idl2tcl* compiler. By adding information into a local Interface Repository, you can remove your dependency on setting up an external Interface Repository in a separate process. See the *Interface Repository* section for more information.

INVOCATIONS

Operations are invoked, and attributes are accessed, using the handle as command. The first parameter to that command is the operation name, and following parameters are arguments to that operation.

```
set handle [corba::string_to_object ...]
$handle ?options? operation ?args ...?
```

operation must be a valid operation for the type of object identified by *handle*, and *args* must match the parameter list of that operation. For **inout** and **out** parameters, the name of a variable must be passed. In the case of **inout**, that variable must contain the value to be sent. In the case of both **inout** and **out**, the variable is set to the value returned by the operation.

The event loop is entered while waiting for the response from the server. The operation's result is then returned as a result from the invocation.

For attributes, use the attribute name for *operation*. Without an argument, the value of that attribute is

returned. With one argument, the attribute is set, and no value is returned.

The following operations are available for all handles:

_is_a *id*
: Returns 1 if the object identified by the handle is or is derived from the Repository Id *id*. Returns 0 if the object identified by the handle is not derived from *id*. Throws an exception if the correct value can not be determined.

_get_interface
: Returns an object reference (as handle) to the **InterfaceDef** object that identifies this type in the Interface Repository. This requires that the server is connected to a well-configured Interface Repository.

_is_equivalent *oh*
: Returns 1 if the object reference identified by the handle is equivalent to the object reference identified by the handle *oh*. Returns 0 if they can be reliably determined to be different. Throws an exception otherwise.

_duplicate
: Returns a new handle for the same object reference. You need to use this e.g. in servers which handle object references. See the *Server Side Scripting* section below.

The following options are accepted:

-async Initiates an asynchronous invocation. Instead of entering the event loop, the invocation returns immediately, and processing of the invocation is handled in the background, within the event loop. Instead of returning the operation's result, an *asynchronous request identifier* is returned. This identifier can subsequently be passed to the **corba::request** to poll or wait for the operation to complete and ultimately be used with the **get** subcommand of **corba::request** to receive the operation's result. Note that the application must update the event loop in order to process asynchronous requests.

-callback *command*
: Again, an asynchronous invocation is initiated, and an asynchronous request identifier is returned. If the invocation has completed, *command* is called at global level with the asynchronous request identifier as single parameter. This callback procedure is expected to call **corba::request get** to receive the operation's result.

LANGUAGE MAPPING

This section describes the mapping of IDL data types to Tcl types.

Primitive Types
: **short**, **long**, **unsigned short**, **unsigned long**, **long long** and **unsigned long long** values are mapped to Tcl's integer type. Errors may occur if a value exceeds the numerical range of Tcl's integer type.

  float, **double**, **long double** values are mapped to Tcl's floating point type.

  string and **wstring** values are mapped to Tcl strings.

  boolean values are accepted as 0, 1, true, false, yes and no. In a result, they are always rendered as 0 (false) and 1 (true).

  octet, **char** and **wchar** values are mapped to strings of length 1.

  fixed values are mapped to a floating-point value in exponential representation. Depending on their scale and value, it may or may not be possbile to use the value in a Tcl expression.

Structs **struct** values are mapped to a list. For each element in the structure, there are two elements in the list -- the first is the element name, the second is the element's value. This allows to easily assign structures from and to associative arrays, using **array get** and **array set**.

Sequences

sequence values are mapped to a list. As an exception, sequences of **char**, **octet** or **wchar** are mapped to strings.

Arrays **array** values are mapped to a list. As an exception, arrays of **char**, **octet** or **wchar** are mapped to strings.

Enumerations

enum values are mapped to the enumeration identifiers (without any namespace qualifiers).

Unions **union** values are mapped to a list of length 2. The first element is the discriminator, or **(default)** for the default member. The second element is the appropriate union member. Note that the default case can also be represented by a concrete value distinct from all other discriminator values.

Object References

Object references are mapped to handles. Nil object references are mapped to the integer value 0 (zero).

Exceptions

exception values are mapped to a list of length one or two. The first element is the Repository Id for the exception. If present, the second element is the exception's contents, equivalent to the structure mapping. The second element may be omitted if the exception has no members. Exception handling is done using Tcl's normal error handling mechanism; they are thrown using **error** and can be caught using **catch**. The convenience operations **corba::throw** and **corba::try** can be used instead (see above).

Value Types

valuetype values are mapped to a list, like **structs**. For each element in the inheritance hierarchy of a **valuetype**, there are two elements in the list -- the first is the element name, and the second is the element's value. An additional member **_tc_** may be present. If present, its value must be a typecode. In an invocation, this member determines the type to be sent. This mechanism allows to send a derived valuetype where a base valuetype is expected. If no **_tc_** member is present, the valuetype must be of the same type as requested by the parameter. In receiving a valuetype, the _tc_ member is always added. A **valuetype** can also be the integer 0 (zero) for a null value. **custom** valuetypes are not supported.

Value Boxes

Boxed **valuetype** types are mapped to either the boxed type or to the integer 0 (zero) for a null value. In the case of boxed integers, the value 0 will always be read as a null value rather than a non-null value containing the boxed integer zero. Shoot yourself in the foot if you run into this problem.

Typecodes

TypeCode values are mapped to a string containing a description of the typecode:

Typecodes for the primitive types **void**, **boolean**, **short**, **long**, **unsigned short** ,**unsigned long**, **long long**, **unsigned long long**, **float**, **double**, **long double**, **char**, **octet**, **string**, **any**, **TypeCode** are mapped to their name.

Bounded string typecodes are mapped to a list of length two. The first element of the list is the identifier **string**, the second element is the bound.

Bounded wstring typecodes are mapped to a list of length two. The first element of the list is the identifier **wstring**, the second element is the bound.

struct typecodes are mapped to a list of length three. The first element is the identifier **struct**. The second element is the Repository Id, if available (else, the field may be empty). The third element is a list with an even number of elements. The zeroth and other even-numbered elements are member names, followed by the member's typecode.

union typecodes are mapped to a list of length four. The first element is the identifier **union**. The second element is the Repository Id, if available (else, the field may be empty). The third element

is the typecode of the discriminator. The fourth element is a list with an even number of elements. The zeroth and other even-numbered elements are labels or the identifier **default** for the default label, followed by the typecode of the associated member.

exception typecodes are mapped to a list of length three. The first element is the identifier **exception**, the second element the Repository Id, and the third element is a list with an even number of elements. The zeroth and other even-numbered elements are member names, followed by the member's typecode.

sequence typecodes are mapped to a list of length two or three. The first element is the identifier **sequence**, the second element is the typecode of the member type. The third element, if present, denotes the sequence's bound. Otherwise, the sequence is unbounded.

array typecodes are mapped to a list of length three. The first element is the identifier **array**, the second element is the typecode of the member type, the third element is the array's length.

enum typecodes are mapped to a list of length two. The first element is the identifier **enum**, the second element is a list of the enumeration identifiers.

Object reference typecodes are mapped to a list of length two. The first element is the identifier **Object**, the second element is the Repository Id of the IDL **interface**.

fixed typecodes are mapped to a list of length three. The first element is the identifier **fixed**. The second element is the number of significant digits, the third element is the scale.

valuetype typecodes are mapped to a list of length five. The first element is the identifier **valuetype**. The second element is the Repository Id. The third element is a list of non-inherited members. For each member, there are three elements in the list, a visibility (**private** or **public**), the member name and the member's typecode. The fourth element is the typecode of the valuetype's concrete base, or 0 (zero) if the valuetype does not have a concrete base. The fifth element is either an empty string or one of the modifiers **custom**, **abstract** or **truncatable**.

Boxed **valuetype** typecodes are mapped to a list of length 3. The first element is the identifier **valuebox**. The second element is the Repository Id, and the third element is the typecode of the boxed type.

A recursive reference to an outer type (in a **struct**, **union** or **valuetype**) can be expressed by a list of length two. The first element is the identifier **recursive**, the second element is the Repository Id of the outer type, which must appear in the same typecode description.

TypeCode values can be constructed manually, or retrieved from the Interface Repository using the **corba::type** command.

Any **any** values are mapped to a list of length two. The first element is the typecode, and the second element is the value.

INTERFACE REPOSITORY

The *Interface Repository* is vital for the operation of **Combat**, and it is important that you understand its importance. **Combat** is fully dynamic and posesses no "compile-time" knowledge of interfaces. This is different from other language mappings (e.g. C++ or Java), where such knowlege is available by the way of an IDL "compiler" in the form of stubs and skeletons.

Combat instead pulls the information from the Interface Repository, at runtime. **You** must take care that an Interface Repository is available and correctly configured. You will have to load the Interface Repository with the IDL files that correspond to your interfaces.

The **Combat** package does not include an Interface Repository, but there is one available for download on the **Combat** home page, from the Mico ORB. Using that Package, you would first start and load the Interface Repository using, on the command line

ird --ior /tmp/ird.ior

idl --feed-ir -ORBInitRef InterfaceRepository=file:///tmp/ird.ior *myidlfile.idl*

On the second command line, replace *myidlfile.idl* with the name of your IDL file. Repeat that command if you have multiple IDL files. To make **Combat** use that Interface Repository, you would then use

corba::init -ORBInitRef InterfaceRepository=file:///tmp/ird.ior

That above was a quick start, but unfortunately, things are more complicated than that, which means that you have more options to choose from. Read on for more detail.

Combat tries to do the following in order to get type information for an interface:

1 **Combat** sends a **_get_interface** request to the server. If the server is connected to a correctly configured Interface Repository, this will return a pointer to a server-side Interface Repository, and **Combat** will then read interface information from there. Unfortunately, few servers are set up this way.

You can try this for yourself by invoking the **_get_interface** operation on a handle manually. If you get a handle in return, all is fine. If you get a nil reference or an exception, this possibility has failed.

2 Next, **Combat** looks for a Repository Id inside the object reference. Unfortunately, CORBA allows the Repository Id field in an object reference to be empty, and the field is always empty for object references of the **corbaloc** kind. If a Repository Id is found, **Combat** contacts its own Interface Repository, which must have been configured as an initial reference by the name of **InterfaceRepository** upon **corba::init**, and tries to look up the Repository Id in it.

You can try this for yourself by acquiring the **InterfaceRepository** initial reference using **corba::resolve_initial_references**, and calling its **lookup_id** operation with the Repository Id as parameter. If you get a handle in return, all is fine. If you get an exception, this possibility has failed.

3 If both strategies above fail, then you have a problem. By the way, one popular reason why the second strategy fails is that the object reference is of a derived type, while only the base type is registered in the Interface Repository (this frequently happens e.g. with the Naming Service, where you load the Interface Repository with the standard CosNaming IDL, but most ORBs actually implement a type that is derived from CosNaming::NamingContextExt). Another frequent reason is using **corbaloc:** object references, which do not contain a Repository Id.

In this case, you can try to force **Combat** into accepting a certain type for that object reference by using the **_is_a** operation:

$handle **_is_a** *id*

Combat will then ask the server whether it is compatible with the Repository Id *id*. If yes, then **Combat** will use *id* as the object reference's type, and look it up as above. In the case of the Naming Service, you would e.g. use

set ns [corba::resolve_initial_references NameService]
$ns _is_a IDL:omg.org/CosNaming/NamingContextExt:1.0

The C++ version of **Combat** also includes a **idl2tcl** utility that distils IDL files into Tcl strings, which can then be used to populate an internal Interface Repository. **idl2tcl** is not included here because of its dependency on the C++ version of Combat and a separate ORB. Its files, however, can also be used with this Tcl version of **Combat**.

SERVER SIDE SCRIPTING

Combat includes a pretty complete implementation of the Portable Object Adapter (POA). You can access the POA by resolving the **RootPOA** initial reference. All operations on the POA follow the normal language mapping rules as above.

A single exception that does not follow the language mapping rules is the third parameter of the **create_POA** operation (*policies*), which for convenience takes a list of policy *identifiers* instead of a list of policy *objects*.

Servants are mapped to [incr Tcl] (or tcl++) objects and must implement a public method **_Interface** that takes no parameters and returns the Repository Id of the interface that the servant implements. Operations are mapped to public methods, and attributes to public variables. Servant classes may optionally inherit from **PortableServer::ServantBase**. This inheritance is not required per se, but only by this do servants inherit the **_this** method.

If your servant has operations that handle object references, then you must be careful about memory management. As you may have noticed, object references are not (cannot be) reference counted as they are in e.g. C++. To avoid memory leakage, the runtime releases all handles that are passed into your operations as **in** or **inout** parameters. If you want to keep a copy, you must duplicate the handle in question using the **_duplicate** method. Also, when you return a handle as result, **inout** or **out** parameter, you must return a duplicate, if you want to keep the original handle for yourself. This rule applies recursively to all object references embedded in other types. The **corba::duplicate** command exists to duplicate all handles embedded in a type.

Note that POA references are not object references, and need not be released by **corba::release**.

The **Cookie** native type is mapped to string.

CODESET SUPPORT

Combat can use about any popular codeset as transmission codeset (*TCS*), thanks to Tcl's excellent support in this area. See the output of **encoding names** for supported codeset encodings. However, a matching entry for each codeset must be present in an internal table that maps the name of the Tcl encoding to the OSF registry values used by CORBA. Currently, this table only includes western codesets (because of the author's heritage and laziness). Contact the author if you want support for other codesets.

EVENT LOOP

Combat relies on fileevents for handling communication. In a synchronous invocation, the event loop is entered until the response from the server is received (internally, **update** is called). When asynchronous requests are used, the application must call **update** at times to allow processing to continue.

In a server, the event loop must be entered to allow reception of incoming requests. Typically, a server application ends in a '**vwait** *forever*' command, where the variable *forever* is never set.

FREQUENTLY ASKED QUESTIONS

All I get is **IDL:omg.org/CORBA/INTF_REPOS:1.0**!

This exception is thrown when **Combat** does not find type information for an object reference. Make sure that an Interface Repository is available, and set up to contain the correct information. Read the *Interface Repository* section above for more information.

I cannot talk to my Naming Service!

You need to configure the initial reference using **-ORBInitRef** (see the documentation for **corba::init** and **corba::string_to_object**).

Most ORBs have proprietary means of locating their own services. For example the JDK ORB is able to find its own Naming Service (*tnameserv*) without any options or configuration efforts by using default, compiled-in object references. Such proprietary information is not available to **Combat**, so you must configure all initial references using **-ORBInitRef**.

Also, in order to use the Naming Service, you will have to load the IDL for the Naming Service into the Interface Repository.

Using **corbaloc:** does not work!

The problem with **corbaloc:** is that such object URLs do not include the Repository Id. Hence, **Combat** will, as its last resort to find out the object reference's type, send a **_get_interface** request to the server, which will likely fail. See the *Interface Repository* section above for more information.

The solution is to give **Combat** a hint about the type by sending a manual **_is_a** invocation with the expected Repository Id as a parameter. **Combat** takes notice of succeeding **_is_a** requests, and can then use that verified Repository Id for usage in the local Interface Repository, e.g.

$handle _is_a IDL:omg.org/CosNaming/NamingContextExt:1.0

in the case of a Naming Service that has been contacted with a **corbaloc:** URL.

I cannot send or receive **wstring**s!

Codeset handling in **Combat** should be pretty correct. However, the encodings for wide strings have been pretty broken in GIOP versions prior to 1.2, and **Combat** refuses to send wstrings over GIOP 1.0 or GIOP 1.1 connections. Check if the remote ORB supports GIOP 1.2 (e.g. by running an iordump utility on its object references). You might want to switch on GIOP debug messages **-ORBDebug GIOP**, for **Combat**'s comments on the situation.

One popular example is the ORB in JDK 1.2 or 1.3, which only supports GIOP 1.1. The ORB in JDK 1.4 should work fine.

TODO

Combat is a reasonably complete ORB. But here are ideas for new features or sensible supporting projects.

Implement the remaining data types (long long, long double, fixed).

IIOP over SSL. Shouldn't be hard with TLS.

Portable Interceptors.

Rewrite **idl2tcl** in Tcl to avoid the dependency on Combat/C++. Would be much easier if there was a grammar package (a la Yacc) for Tcl.

Contact the author if you are interested in working on any of these projects, or in funding them.

SEE ALSO

The **Combat** homepage, http://www.fpx.de/Combat/

KEYWORDS

CORBA, IIOP, GIOP, Object Request Broker