CORBA Scripting with Tcl
Version 0.8

Frank Pilhofer

November 29, 2008

Abstract

Combat is a CORBA Object Request Broker that allows the implementation of CORBA clients and servers in the Tcl programming language.

On the client side, Combat is not only useful to easily test-drive existing CORBA servers, including the ability for rapid prototyping or to interactively interface with servers from a console, but makes Tcl an exciting language for distributed programming. Also, Tk allows to quickly develop attractive user interfaces accessing CORBA services. Server-side scripting using [incr Tcl] classes also offers a wide range of possibilities. Applications can be trivially packaged as Starkit or Starpack for easy cross-platform deployment.

Combat is compatible with the CORBA 3.0 specification including the IIOP protocol, and has been tested to interoperate with a wide range of open-source and commercial ORBs, including Mico, TAO and ORBexpress.

Combat is written in pure Tcl, allowing it to run on all platforms supported by Tcl, which is a much wider range than supported by any other ORB.

Please visit the Combat homepage, `http://www.fpx.de/Combat/` for the latest information about Combat.

Contents

1 Introduction

1.1 About This Document

This document is part user manual, part reference manual. As a result, some sections are more verbose than others. It is arranged so that it could be read from top to bottom without too many forward references. The reader may want to make a first quick pass, skipping over the "reference" sections, returning to them as necessary.

This document is neither a CORBA nor Tcl tutorial. The reader is assumed to be familiar with the Tcl programming language and should have a basic understanding of CORBA concepts. Prior exposure to CORBA software development is recommended.

1.2 Quick Start

Let's begin with an example of a common Combat use case, accessing an existing CORBA-based server. This assumes the following:

- The server is running.
- You have the server's stringified object reference, i.e., the "IOR:" string.
- You have the IDL file that defines the interface that the server implements.

For example, consider the following IDL file, in which the server implements the "Bank" interface. The Bank supports the `create` operation to open a new account. An account, in turn, is an object that supports the `deposit`, `withdraw` and `balance` operations.

```
interface Account {
  void deposit (in unsigned long amount);
  void withdraw (in unsigned long amount);
  long balance ();
};
interface Bank {
  Account create (in string name, in string passwd);
};
```

Without getting into all the details, the first step is to process the IDL file into type information that Combat can use. This is done using the "idl2tcl" application. (For more details about the idl2tcl application, see section 6.) Given the file "account.idl", you would run the following command in a console to produce the file "account.tcl:"

```
idl2tcl account.idl
```

With this preparation, you can launch a Tcl shell, load the Combat ORB, and load the type information for the above IDL file that we just generated:

```
package require combat
source account.tcl
```

At this point, we are ready to connect to the server. This is done by passing the server's stringified object reference to the `corba::string_to_object` method, just like in any other programming language. (For more details about this command, see section 2.3.1.) You can copy&paste the "IOR:" string, or you could read it from a file with a few extra lines of code.

```
set Bank [corba::string_to_object IOR:...]
```

The "Bank" varaible now holds an *object reference* that we can use to exercise the server using its IDL interface. Calling the bank's "create" operation returns an object reference for an "Account" object, which supports the "deposit," "withdraw," and "balance" operations.

```
set Account [$Bank create MyName MyPassword]
$Account deposit 700
$Account withdraw 450
puts "Current balance is [$Account balance]."
```

This simple but complete example glanced over some of the details but should give you a good idea of how easy it is to develop code using Combat. Feel encouraged to follow the above steps to interact with some of your existing CORBA servers. You will find that the ability to "chat" with a live server from a Tcl console is a powerful prototyping and debugging tool.

A slightly more elaborate version of the above example can be found in the "demo/account" subdirectory. For a more well-documented example, see section 7.

1.3 Features

Combat has the following features:

- IIOP/GIOP 1.0, 1.1 and 1.2 (unidirectional).
- Straightforward IDL to Tcl mapping that supports all data types, including TypeCode, Any, Objects by Value and recursive data types.
- Asynchronous invocations, callbacks and timeouts.
- Support for the IOR:, corbaloc:, corbaname:, file: and http: stringified object reference formats.
- Server-side scripting with full Portable Object Adapter (POA) support.
- Codeset negotiation (when using GIOP 1.2). Thanks to Tcl's encoding system, a wide range of character sets is supported.
- Download of type information at runtime, using an Interface Repository or from a server that supports the CORBA Reflection specification.
- Fully event based, the event loop is kept running while waiting for a server response.

1.4 Glossary

These terms are essential for Combat development.

Object Reference A Tcl command with the same interface as the server object as defined in the IDL description. Operations (or the getting/setting of attributes) on the object requence transparently cause a server invocation to happen, i.e., a request is sent to the remote server, and the response from the server is processed and returned. Object references come into existing using the `corba::string_to_reference` command or as a result from a method invocation. Object references are usually stored in variables. Object references are the equivalent to interface "_ptr" types when programming CORBA in C++. Object references can be duplicated using `corba::duplicate` and must eventually be released using `corba::release`.

Stringified Object Reference Also known as IOR (for Interoperable Object Reference), a stringified object reference is a *string* that uniquely identifies a server object (servant). No operations can be performed upon an IOR directly; it must first be incarnated into an *object reference* using `corba::string_to_reference`. Stringified object references usually begin with the letters "IOR:", "corbaloc:" or "corbaname:".

Local Object Like an object reference, a local object is a Tcl command with the same interface as the object it represents. Invocations on a local object do not cause requests to be sent to a remote servant, but are processed locally. The POA, POAManager and POACurrent are used via local objects. A local object is not associated with a stringified object reference.

Servant A servant implements the state and behavior that is associated with a CORBA object. They are the target of CORBA remote method invocations. In Combat, a servant is an instance of an [incr Tcl] class that inherits directly or indirectly from `PortableServer::ServantBase`. Servants are *activated* with the Portable Object Adapter (POA), which creates object references that clients can use to interact with the servant.

1.5 Interface Repository

Because it posesses no "compile-time" knowledge of object interfaces, the Interface Repository is vital for the operation of Combat. This is different from other language mappings, where such knowledge is generated by the IDL "compiler" and eventually linked into the application. Combat instead pulls type information from Interface Repositories, at runtime. Before an interface can be used (either as a client or a server), its type information must be loaded into the Interface Repository. Before accessing a service, Combat must learn its type.

Most commonly,

- Combat knows the service's type (its "Repository Id") because it is contained in an object's address, and
- Combat knows the type's interface information because it was generated from the service's IDL file using the "idl2tcl" application (which generates a ".tcl" file) and loaded into Combat's local Interface Repository using Tcl's "source" command.

Alternatively, Combat can also download interface information from local or remote CORBA Interface Repositories, or even from the service itself, if its ORB supports the CORBA Reflection specification.

The most frequent caveat is that a stringified object reference does not necessarily contain the service's Repository Id that Combat depends on. This typically applies to stringified object references of type "corbaloc." In this case, you might need to apply workarounds described in section 8.

See section 6 for more information about the "idl2tcl" application.

2 ORB Operations

2.1 Initialization & Features

2.1.1 corba::init

Before any of the other commands can be used, the ORB should be initialized. This is performed using the `corba::init` command.

`corba::init ?`*`parameters`*`?`

All ORB-specific parameters (see below) are consumed, the remaining parameters are returned. It is common practice to pass an application's command-line arguments to `corba::init`, i.e.,

```
set argv [eval corba::init $argv]
```

Afterwards, `argv` contains the remaining options. This allows users of the application to pass ORB parameters (such as initial references) on the application's command line.

Use of this command is optional. If `corba::init` is not called explicitly, it is implicitly called with an empty parameter list when the ORB is first used.

Combat accepts the following ORB-specific parameters:

-ORBInitRef *name*=*value* Sets the initial reference *name* (see `corba::resolve_initial_references`) to *value*. The *value* must be a stringified object reference.

-ORBDefaultInitRef *value* Sets the default initial reference. See `corba::resolve_initial_references` for more information.

-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 exceeding available memory. If set, GIOP messages whose size (in octets) exceed *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. The *value* must be a number; it may be followed by the "k", "m" or "g" character to indicate kilobytes, megabytes or gigabytes, respectively.

-ORBConnectionIdleTimeout *value* If a connection to a remote ORB remains idle for this period, it is closed to conserve system resources. The connection will be transparently reestablished upon the next request. Connections are never closed if there are outstanding requests (e.g., if a server takes a long time to process a request). The *value* is specified in milliseconds. The default is 3600000 (one hour).

-ORBSendCancelRequestMessage *value* Controls whether a GIOP CancelRequest message is sent to the server when a request is cancelled (using `corba::request cancel` or because it timed out). The CORBA compliant default is to send a CancelRequest message in this case, but some ORBs do not support this message and may close the connection. To avoid issues when interoperating with such ORBs, set this option to false.

-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 once the RootPOA is accessed for the first time.

-ORBHostName *name* The host name that will be used in object references to refer to the local host. If omitted, the local host's default name (i.e., the name returned from [info hostname]). In environments where DNS is not available, it may be necessary to use this option with the local host's IP address.

-ORBServerId *value* The name that a persistent POA encodes in IORs to distinguish this server from others. Also, if a persistent POA is created with the same name, and if an object is activated in this POA with the same object id, then its entire object key will be *value*. The object will thus be accessible using the URI corbaloc::*host*:*port*/*value*, where *host* is the local host name (or the string passed to -ORBHostName) and *port* is the automatically chosen server port number (or the number passed to -ORBServerPort).

-ORBNativeCodeSet *value* Sets the native codeset to be used and advertised as SNCS-C ("server native codeset for char"; see the CORBA specification for more details). If omitted, the native codeset is determined using [encoding system]. *value* can be an OSF registry value or Tcl encoding name.

-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), *poa* (Portable Object Adapter operations) and *all* (for all of them). This debug output may be useful in debugging interoperability issues.

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

2.1.2 corba::feature

Syntax:

```
corba::feature names
corba::feature require ?-exact? feature ?version?
```

The `names` subcommand returns a list of feature tokens supported by this implementation. For Combat, this list is *core*, *async*, *callback*, *type*, *poa*, *register*, *dii* and *combat::ir*.

The `require` subcommand has three possible usages. If called with only a feature token, it succeeds if that feature is implemented. If that feature is not available at all, an error is returned. If a version number is mentioned, the command only succeeds if the implementation provides that feature with the same major number and at least the same minor number. With the `-exact` option, the given version number must match the implemented version exactly.

It is expected that a feature is upwards compatible within the same major version number, i.e., version 1.3 supports all operations that version 1.1 did, but version 2.1 is probably incompatible. Major version 0 is an exception in that it does not assume backwards compatibility.

As long as the Tcl language mapping is not official, Combat reports the version number of the supported features to be the same as the Combat version (i.e., less than 1.0).

2.2 Initial References

2.2.1 corba::resolve_initial_references

Syntax:

```
corba::resolve_initial_references id
```

Obtains an initial reference. Depending on *id*, returns an object reference or a local object. Valid ids include "RootPOA", "POACurrent", "CodecFactory" (all implemented as local objects), all ids registered using the -ORBInitRef parameter to the `corba::init` command, and all ids registered using the `corba::register_initial_reference` command.

If a default initial reference was set using the -ORBDefaultInitRef parameter to the `corba::init` command, and id is not a registered initial reference, then the default initial reference is concatenated with *id*; the resulting string is passed to `corba::string_to_object`.

2.2.2 corba::list_initial_services

Syntax:

```
corba::list_initial_services
```

Returns a list of ids that can be used with `corba::resolve_initial_references`. Note that the "RootPOA" id is not listed until the Root POA is first used.

2.2.3 corba::register_initial_reference

Syntax:

```
corba::register_initial_reference name obj
```

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

2.3 Object Reference Operations

2.3.1 corba::string_to_object

Syntax:

```
corba::string_to_object string
```

Interprets the *string* as a stringified object reference, and converts it to an object reference. The *string* may have any of the following formats:

IOR: This is the classic CORBA format for stringified object references, a very long string with the "IOR:" prefix and the remainder in hexadecimal data.

corbaloc::*host*[:*port*]/*ObjectKey* This format contains the server's IP address, its port number, and the object key of the target object. If *port* is omitted, the 2089 is used as default.

corbaname::*host*[:*port*][/*ObjectKey*][#*name*] This format references an entry in the Naming Service. The *host*, *port* and *ObjectKey* identify a Naming Context. If *port* is omitted, 2089 is used as default. If *ObjectKey* is omitted, "NameService" is used. *Name* identifies an entry in the Naming Service. This entry is read and returned. If *name* is omitted, the reference of the Naming Context itself is returned.

file://[*host*]*path* Reads a stringified object reference from the given file. *host* should be the empty string or be an alias of the local host. *path* is an absolute path name. The contents of this file are read and then passed to `corba::string_to_object`. As a special case, on Windows, if *host* is the empty string, then *path* may start with a drive name followed by a colon.

http:// The referenced URL is downloaded (requires the http package). Its contents are then passed to `corba::string_to_object`.

2.3.2 corba::object_to_string

Syntax:

```
corba::object_to_string ref
```

Converts the object reference *ref* into a stringified object reference, using the "IOR:" format.

2.3.3 corba::release

Syntax:

```
corba::release ref
corba::release typecode value
```

Releases the memory associated with object references. All object references must eventually be released using this function to avoid memory leaks.

When called with a single parameter, releases the object reference *ref*.

When called with two parameters, the first parameter is a typecode, and the second parameter is a value that maches the typecode. This version can be used with complex data structures to release all object references that are members of the data structure.

2.3.4 corba::duplicate

Syntax:

```
corba::duplicate ref
corba::duplicate typecode value
```

Duplicates an object reference. This is usually done before passing object references to a function that eventually releases its object references, so that the duplicate can be used independently.

When called with a single parameter, returns a new object reference that is a duplicate of *ref*.

When called with two parameters, the first parameter is a typecode, and the second parameter is a value that matches the typecode. This version can be used with complex data structures. A copy of value is returned, with all object referencesthat are members of the data structure duplicated.

2.4 Interface Repository Operations

2.4.1 corba::const

Syntax:

```
corba::const id
```

Looks up *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 using type any (i.e., a list of the constant's typecode and value).

2.4.2 corba::type

Syntax:

```
corba::type cmd ?args?
```

Handles type definitions. Can be used to ensure type safety. Its usage depends on the subcommand:

corba::type of *id* Looks up *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.

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

corba::type 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).

2.4.3 combat::ir

Syntax:

```
combat::ir add args
```

This command is used to load interface information into the local Interface Repository. *args* must be a string which is the result of processing an IDL file with the "idl2tcl" application.

There is normally no need to call this function manually, it is implicitly called when loading a file generated by "idl2tcl" using the "source" command.

3 Client Side Scripting

3.1 Invocations

3.1.1 Calling Operations and Attributes

As already noted, an object reference is a Tcl command same interface as the server object as defined in the IDL description. Using the object reference results in a remote method invocation. The generic format for method invocations and attribute access is:

```
ref ?options? operation ?args?
```

ref must be a valid object reference. The *operation* must be a valid operation or attribute according to the service's interface.

For operations, *args* has the same number of parameters as in the IDL interface description. "in" parameters are passed *by value*, as expected, while "out" and "inout" parameters are passed *by reference*, i.e., by passing the name of a variable that contains a value (in the case of "inout" parameters) and/or will receive the output parameter (in the case of "inout" and "out" parameters). The operation's result is returned.

For attributes, *args* can be empty to read the attribute. In this case, the value of the attribute is returned. To set the attribute, pass the new value as *args*. This returns an empty value.

Valid *options* are "-async", "-callback" and "-timeout". When used without the "-async" or "-callback" options, two-way invocations are synchronous and will wait for the server reply before returning. One-way operations are always asynhronous. See section 3.2 for more information about asynchronous invocations.

The "-timeout" option takes a single parameter, which is the timeout in milliseconds. If a reply is not received within this period, then the request fails with the IDL:omg.org/CORBA/TIMEOUT:1.0 exception.

Note that the ORB releases any object references that are passed to "inout" parameters.

See section 5 below for details about how CORBA data types are mapped to Tcl.

3.1.2 Brief Example

For example, consider the following IDL:

```
interface A {
  short foo (in long val, inout short flags, out string name);
  attribute string bar;
};
```

Now, assuming that the variable "aref" contains an object reference of type A, the following Tcl code could be used to invoke the "foo" operation and to access the "bar" attribute:

```
set flags 42
set res [$aref foo -1 flags name]
puts "Bar = [$aref bar]"
$aref bar "Hello World"
```

The second line passes -1 as the "val" parameter and 42 as the "flags" parameter. After returning from the "foo" operation, "flags" and "name" will be updated to contain the values returned from "foo." The third line returns the current value of the "bar" attribute, while the fourth line sets the attribute to a new value.

3.1.3 Built-In Operations

An object reference also supports the following "built-in" operations which have the same semantics as defined in the CORBA specification.

`_get_interface` Queries the servant for its interface. Returns an object reference of type `CORBA::InterfaceDef`. This requires that the remote service is connected to a properly set-up Interface Repository. Note that use of the returned object reference requires that the IDL for the Interface Repository has been loaded into the Interface Repository.

`_is_a` ***type*** Takes a type, which may be specified as a Repository Id or an absolute scoped name as parameter and returns true (1) if the object implements the given interface. If Combat did not have type information for this object reference, if type information for *repoid* is available in the Interface Repository, and if this operation returns true, then Combat will remember *repoid* as this object reference's type. This feature can be used as a workaround if an object reference, e.g., one of "corbaloc:" style, does not contain type information. Scoped names can only be used for types for which information is available in the Interface Repository.

`_non_existent` Returns true (1) if the server providing the implementation for this object has ceased to exist. A false return value (0) does not guarantee that any following invocations will succeed.

`_is_equivalent` ***ref*** Takes another object reference as parameter and returns true (1) if the objects referenced by both object references are equivalent, or false (0) if not. This operation is implemented locally.

`_duplicate` Returns a duplicate of the object reference. This is equivalent to *corba::duplicate*. This operation is implemented locally.

Note that there is no need for an "is_nil" operation, because nil addresses are mapped to the value 0 rather than object references.

3.2 Asynchronous Invocations

3.2.1 Calling Operations and Attributes Asynchronously

The "-async" and "-callback" options can be used to make an invocation asynchronous.

```
ref -async operation ?args?
ref -callback cb operation ?args?
```

In either case, the call returns immediately. Instead of the operation's result, an *async-id* is returned. `The -callback` option arranges for the given callback procedure *cb* to be called when the reply is received from the server. The procedure *cb* is called at global level with the async-id as single parameter. It is guaranteed that the callback is not executed before the above call returns.

3.2.2 corba::request

The `corba::request` command exists to monitor the status of asynchronous invocations in progress.
Syntax:

```
corba::request get async-id
corba::request poll ?async-id ...?
corba::request wait ?async-id ...?
corba::request cancel async-id ...
```

`get` Waits until the asynchronous request with the given *async-id* has completed, and returns the result of the operation, or throws an exception in case of a failure. This also extracts any "out" or "inout" parameters *within the context of the* `get` *call* (see below).

`poll` If called without a parameter, it checks if any of the outstanding asynchronous invocations have completed. If `poll` is called with one or more *async*-id, it checks if any of these asynchonous invocations have finished. If yes, a single async-id is returned. `corba::request get` should then be called for that async-id to retrieve the result. If none of the (given) requests have completed, `poll` returns immediately with an empty result.

`wait` Similar to `poll`, but waits (enters the event loop) until one request has finished and then returns its async-id. If called without arguments, it considers all outstanding requests. If there are no outstanding requests, it immediately returns with an empty result.

cancel Cancels the asynchronous requests with the given *async-ids*. All information about the requests is discarded. `corba::request get` shall not be called. If a callback was configured for this request, it is *not* called.

3.2.3 Callback Procedures

A callback procedure receives an async-id as single parameter and is expected to (directly or indirectly) perform a `corba::request get` for that async-id. Here's a simple example for a callback:

```
proc cb {id} {
  set res [corba::request get $id]
  puts "Result is: $res"
}
```

Note the behaviour with respect to an operation's `out` or `inout` parameters. When sending a request, only the *names* of the variables that were given for the "out" or "inout" parameter are stored, and these variable names are written to in the context in which the corresponding `corba::request get` is executed. So unless global variable names are used (i.e., variable names starting with "::" or those declared "global"), they will be visible in the context of the callback only.

3.2.4 Notes

- The `-async` and `-callback` flags can be used likewise on operations and attributes.
- Ordering is not guaranteed for asynchronous invocations, not even on the same object.
- Asynchronous invocations are only processed in Tcl's event loop.
- If a timeout was specified for an asynchronous invocation, then upon expiration of the timeout, the associated async-id becomes ready, the callback is called if specified, and corba::request get will raise the IDL:omg.org/CORBA/TIMEOUT:1.0 exception.

3.3 Dynamic Invocations

Calling operations and accessing attributes as described above requires that type information is available in the Interface Repository. If type information is not available, a *dynamic invocation* can be used to call operations and access attributes. For a dynamic invocation, type information must be provided "manually." This can be useful for generic programming, if type information becomes available at run-time only.

Syntax:

```
corba::dii ?options? ref spec ?args?
```

The *ref* and *args* parameters are the same as above. *spec* must be 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 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 the parameter direction, i.e., either "in", "out" or "inout," and the second element is the parameter's typecode. The fourth element of spec is a list of exception typecodes that this operation may throw.

Valid options are "-async", "-callback" or "-timeout", as above.

4 Server Side Scripting

4.1 Implementing Servants

Like other ORBs, Combat supports the Portable Object Adapter (POA) and its associated APIs. Implementing servants is therefore very similar as in other programming languages.

Servants are realized using [incr Tcl] classes that inherit, directly or indirectly, from the base class PortableServer::ServantBase. The implementation must provide public variables for IDL attributes and public methods for IDL operations. Methods follow the same mapping rules as for client-side scripting, including the by-reference rules for "inout" and "out" parameters.

Note that the ORB releases all object references that are passed into and out of an operation. Use corba::duplicate to preserve object references beyond the method invocation.

Because Combat does not have compile-time type information, run-time type information must be supplied. This is done by implementing the public method "_Interface" (with a leading underscore and a capital I). This method must not take any parameters and must return the type of the most-derived IDL interface that the servant supports, either as a Repository Id or as an absolute scoped name. Type information for this interface must be available in the local Interface Repository. Like with client-side scripting, this is achieved by processing the IDL file with the "idl2tcl" application and loading the resulting ".tcl" file using the "source" command.

It is the application's responsibility to create and delete servant instances using the normal [itcl Tcl] semantics. The application must not delete any active servants.

Servants are not immediately accessible from the outside after their creation. They must first be *activated* using the Portable Object Adapter.

Servants inherits the "_this" methods, which works the same as in the C++ mapping:

1. Within the context of a request invocation, "_this" returns a new object reference for the servant.

2. Outside the context of a request invocation, if the servant has not yet been activated, and if its POA has the IMPLICIT_ACTIVATION policy, the servant is activated, and an object reference for the newly-activated servant is returned.

3. Outside the context of a request invocation, if the servant has already been activated, and if its POA has the UNIQUE_ID policy, an object reference for the already-active servant is returned.

4.2 Implementation Example

Considering the following IDL file:

```
interface HelloWorld {
  attribute long messageCounter;
  void hello (in string message);
};
```

Its implementation could look like:

```
itcl::class HelloWorld_impl {
  inherit PortableServer::ServantBase
  public method _Interface {} {
    return "::HelloWorld"
  }
  public variable messageCounter 0
  public method hello {message} {
    puts "The client says: $message"
    incr messageCounter
  }
}
```

Here is a second example to demonstrate usage of "out" and "inout" parameters. As with method invocations, "in" parameters are passed by value, while "out" and "inout" parameters are passed by reference. Considering the following IDL file:

```
interface A {
  short op (in long val, inout short flags, out string name);
};
```

The implementation of the "op" method receives variable names for the "flags" and "name" parameters. The implementation can then "import" these variables using Tcl's "upvar" command. Therefore, an implementation for the above interface could look like:

```
itcl::class A {
  inherit PortableServer::ServantBase
  public method _Interface {
    return "::A"
  }
  public method op { val flags_name name_name } {
    # The next line imports ``flags'' and ``name''
    upvar $flags_name flags $name_name name
    puts "val is $val"
    puts "flags is $flags"
    set flags -1
    set name "Hello World"
    return 42
  }
}
```

4.3 The POA Local Object

A local object for the Root POA is obtained using corba::resolve_initial_references:

```
set rootPOA [corba::resolve_initial_references RootPOA]
```

POA local objects support all operations as defined in the CORBA specification. The usual type mapping rules apply, with a single exception. The second parameter to the `create_POA` method is a list of policy *values* rather than a list of policy *objects*. (Note: This obsoletes corresponding factory operations like `create_lifespan_policy`.)

For example:

```
set myPOA [$rootPOA create_POA 0 {USE_SERVANT_MANAGER PERSISTENT}]
```

This creates a new POA as a child of the Root POA. A new POA Manager is created, because a nil value rather than a local object for an existing POA Manager is passed as the first parameter. The new POA will support persistent objects and use a servant manager.

The "native" data types from the POA specification are represented in the following way:

PortableServer::Servant Servants are instances of an [incr Tcl] class that derives from PortableServant::-ServantBase, as described above.

PortableServer::ObjectId ObjectIds are mapped to Tcl strings.

PortableServer::ServantLocator::Cookie Cookies are mapped to Tcl strings.

4.4 The POA Current Local Object

A POA Current local object can be obtained using corba::resolve_initial_references. A POA Current local object implements all operations as defined in the CORBA specification:

`get_POA` In the context of a method invocation on a servant, returns the POA in whose context it is called.

`get_object_id` In the context of a method invocation, returns the Object Id identifying the object in whose context it is called.

4.5 The POA Manager Local Object

A POA Manager local object is obtained using the "the_POAManager" method on a POA local object. It implements the following methods as defined in the CORBA specification:

`activate` Switches all associated POAs to the "active" state so that they begin serving requests.

`hold_requests` ***wait_for_completion*** Switches all associated POAs to the "holding" state, so that incoming method invocations are queued. Queued requests are delayed until the POA again enters the active state.

`discard_requests` ***wait_for_completion*** Switches all associated POAs to the "discarding" state, so that incoming method invocations are discarded rather than processed.

`deactivate` ***etherealize wait_for_completion*** Switches all associated POAs to the "inactive" state. If *etherealize* is true, a servant manager, if available, is asked to "etherealize" active objects.

4.6 Limitations

Because [incr Tcl] does not support virtual inheritance, Combat does not support multiple inheritance. Single implementation inheritance works, so the implementation for a derived class can inherit from the implementation of the base class (and will thus indirectly inherit PortableServer::ServantBase).

5 The IDL to Tcl Mapping

5.1 Mapping of Data Types

This section describes in detail how IDL data types are mapped to Tcl types.

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

Values of type `float`, `double`, `long double` are mapped to Tcl's floating point type.

Values of type `string` and `wstring` are mapped to Tcl strings.

Values of type `boolean` may take any of the forms for which "string is boolean" returns true. In a result, they are always rendered as 0 (false) and 1 (true).

Values of type `octet`, `char` and `wchar` values are mapped to strings of length 1.

Values of type `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.

Struct Types Values of type `struct` 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`.

Example: the IDL type

```
struct A {
  unsigned long B;
  string C;
};
```

can be matched by the Tcl list "{B 42 C {Hello World}}".

Sequences Values of a `sequence` type are mapped to a list. As an exception, sequences of `char`, `octet` and `wchar` are mapped to strings.

Example: the IDL type

```
typedef sequence<A, 2> D;
```

(following the above example for a structure) can be matched by the Tcl list "{{B 42 C {Hello World}}}". Note the extra level of nesting compared to the struct above.

Arrays Values of `array` type are mapped to a list. As an exception, sequences of `char`, `octet` and `wchar` are mapped to strings.

Enumerations Values of `enum` type are mapped to the enumeration identifiers (without any namespace qualifiers).

Example: the IDL type

```
enum E {F, G, H};
```

can be matched by the Tcl string "G".

Unions Values of a `union` type 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 case labels.

Object References Non-nil object references are mapped to Tcl commands. Nil object references are mapped to the integer value 0 (zero).

Exceptions Values of an `exception` type 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.

Value Types Values of a `valuetype` type 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 with the name "`_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. When receiving a valuetype, the "`_tc_`" member is always added. The value 0 (zero) can be used for a null value.

Note that this language mapping disallows valuetypes that contain themselves (such as graphs with circles).

`custom` valuetypes are not supported.

Value Boxes Boxed `valuetype` values 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 Values of type `TypeCode` 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.
- Typecodes for a `struct` type 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.

- Typecodes for a `union` type 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.
- Typecodes for an `exception` type 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.
- Typecodes for a `sequence` type 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.
- Typecodes for an `array` type 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.
- `Typecodes for an enum` type 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`.
- Typecodes for a `fixed` type 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.
- Typecodes for a `valuetype` 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`.
- Typecodes for a boxed `valuetype` 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.

Examples for legal TypeCodes are:

- `struct {} {s short ul {unsigned long} Q string}`
- `enum {A B C}`
- `union {} short {0 boolean (default) string}`
- `struct IDL:S:1.0 {foo {sequence {recursive IDL:S:1.0}}}`

See the description of corba::type in section 2.4.2. The "of" subcommand can be used to retrieve TypeCode information from the Interface Repository, the "equivalent" subcommand can be used to check TypeCode values against known types.

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

5.2 Exceptions

Exceptions are mapped to Tcl errors. The `corba::throw` command is syntactic sugar for Tcl's native "error," and `corba::try` can be used to handle exceptions thrown with `corba::throw` as well as errors raised with "error."

5.2.1 corba::throw

Exceptions can be thrown with the `corba::throw` command.

Syntax:

```
corba::throw exception
```

The *exception* value must be of exception type according to the type mapping, i.e., a list of length one or two. The first element of the list is the exception's Repository Id or absolute scoped name. The second element is a list of the exception's members according to the mapping for structures. E.g., if the following two exceptions were declared in interface A:

```
interface A {
  exception EX {
    long value;
    string reason;
  };
  exception OOPS {};
};
```

Then these would be valid uses of `corba::throw`:

```
corba::throw {IDL:A/EX:1.0 {value 42 reason "oops, what's up, doc?"}}
corba::throw ::A::OOPS
```

In the second example, the second list element could be omitted because the exception does not have any members.

If this command is used in a servant in the context of a server invocation, and if the exception is not caught within the servant, it is passed back to the client side. If an exception is not caught within a client, the client prints an error message and terminates.

5.2.2 corba::try

The `corba::try` command implements Java-style processing of exceptions and other Tcl errors.

Syntax:

```
corba::try block ?catch {type ?var?} c-block? ... ?finally f-block?
```

First, *block* is evaluated. If a CORBA exception or Tcl error has occured, then the `catch` clauses are searched left to right. Each `catch` clause is associated with a specific exception type and a code block. For the first clause whose *type* matches the exception's type, the associated *c-block* is executed. The *type* can be specified as a Repository Id or as an absolute scoped name.

The special value "..." can be used for *type* to match all CORBA exceptions and Tcl errors. If a variable name *var* is present in a `catch` clause, this variable is set to the exception that has occured. Regardless of whether an exception or error has occured and whether an exception or error has indeed been handled by a `catch` clause, the *f-block* associated with the `finally` clause is, if it exists, always executed. If there are no `catch` clauses, an implicit clause that catches "..." is used. The return value of the `corba::try` statement is computed as follows, in order of priority:

- If a `finally` clause exists and its *f-block* completes with a return value different from `TCL_OK` (i.e. causes itself an error or executes a Tcl `return`, `break` or `continue` statement), then this return code is used.
- If a CORBA exception or Tcl error occurs while executing *block*, and this exception or error is handled by a `catch` clause, then the return value of the associated *c-block* is used.
- If a CORBA exception or Tcl error occurs while executing *block*, and this exception is not handled by a `catch` clause, then this error is used.
- If no CORBA exception or Tcl error occurs while executing *block*, then its return code is used.

One effect of this return value handling is that all code blocks may execute a Tcl `return`, `break` or `continue` statement, which will then be correctly passed along to the surrounding code.

One caveat is that it is not possible to collectively catch all CORBA system exceptions as "CORBA::-SystemException."

5.2.3 Exception Example

Here's a small code example demostrating exception handling:

```
corba::try {
  ...
} catch {::A::EX oops} {
  # The variable oops contains A::EX data
} catch {::CORBA::COMM_FAILURE} {
  # The remote server may be down
} catch {... oops} {
  puts "oops: unexpected exception: $oops"
}
```

6 The idl2tcl Application

6.1 Introduction

As already mentioned above, Combat requires that type information is loaded into the local Interface Repository. The "idl2tcl" application is provided to read IDL files and to produce a Tcl file that can be loaded with the "source" command to feed the IDL file's type information to the local Interface Repository.

idl2tcl works by preprocessing and parsing the IDL file into a CORBA Interface Repository. Because Combat neither implements an IDL parser nor a CORBA-compliant Interface Repository, idl2tcl must rely on a third-party ORB to provide both. Unfortunately, the syntax for loading IDL files into an Interface Repository is not standardized. idl2tcl has built-in knowledge how to accomplish this using the MICO and TAO ORBs, and has command-line options to work with other ORBs.

6.2 Usage

idl2tcl is a command-line application with the following syntax::

```
idl2tcl ?options? idl-file ...
```

Each IDL file is preprocessed and loaded into an Interface Repository. The Interface Repository is started from scratch, i.e., before the first IDL file is loaded, it is empty. Type information for the entire contents of the Interface Repository is then written to a Tcl file in the current directory with the same base name as the first IDL file and the ".tcl" extension. This Tcl file can then be read with the "source" command to load the type information from all IDL files into Combat's Interface Repository.

idl2tcl supports the following options:

--name *name* Names the output file, instead of using the first IDL file's base name. The ".tcl" extension is appended to *name*.

-I*path* This option is passed to the IDL compiler.

-D*name*[=*value*] This option is passed to the IDL compiler.

--impl Writes an implementation skeleton file to the file *name*_skel.tcl, where *name* is the base name of the first IDL file name or the parameter to the "--name" option. This skeleton can then be filled in to ease the effort of implementing servants. If the output file exists, it is not overwritten.

--ir *ior* Process the contents of the CORBA Interface Repository identified by the stringified object reference *ior*. No IDL files are processed. The "--name" option must be supplied.

--ifr *cmd* Use *cmd* to start the Interface Repository. *cmd* must include appropriate command-line parameters so that the Interface Repository's stringified object reference gets written to the file "ir.ior" in the current directory.

--idl *cmd* Use *cmd* to load IDL files into the Interface Repository. To *cmd*, idl2tcl appends the "-ORBInitRef" parameter to set the "InterfaceRepository" initial reference, the set of IDL compiler options ("-I" and "-D" above), and the name of a single IDL file. *cmd* shall return a zero exit status in case of success or non-zero in case of failure, and not write to standard error except in case of error.

-v Be more verbose. This option can be used multiple times for increased verbosity.

6.3 ORB Configuration

As mentioned above, idl2tcl requires an IDL compiler and CORBA Interface Repository from a third part ORB. idl2tcl has built-in knowledge of MICO and TAO:

- To have idl2tcl use MICO, the "ird" and "idl" applications must be available in the search path.
- To have idl2tcl use TAO, the "IFR_Service" and "tao_ifr" applications must be available in the search path.

There are two options to use another ORB's IDL compiler and Interface Repository:

- Use idl2tcl's "--ifr" and "--idl" options, providing the ORB-specific command lines for the IDL compiler and Interface Repository.
- Manually start the Interface Repository, manually load the IDL file(s) into the Interface Repository, and then use idl2tcl's "--ir" option to process this Interface Repository.

6.4 Troubleshooting

This section notes some common issues with idl2tcl:

- The tool that loads IDL files into the Interface Repository generally requires a C preprocessor to be available in your search path. (E.g., MICO's "idl" uses "mico-cpp," while TAO's "tao_ifr" uses a preprocessor that's defined at compile-time.) The search path must bet set so that the required C preprocessor is found.
- Some IDL compilers use built-in search paths to locate standard IDL files (such as "corba.idl"). When you are using the pre-built idl2tcl package on Windows, which is based on MICO, but otherwise use a different ORB for development, it may be necessary to supply the appropriate include path to idl2tcl..

7 Complete Example

This document has already included a number of code fragments. This chapter walks you through a simple but complete client/server application. The code can also be found in the "demo/hello" subdirectory.

7.1 The IDL File

The IDL file, hello.idl, defines the following interface:

```
interface HelloWorld {
  void hello (in string message);
  attribute long messageCounter;
};
```

This file can then be "compiled" with the idl2tcl application:

```
idl2tcl hello.idl
```

This results in the file "hello.tcl".

7.2 The Client

The client is implemented as a command-line application with the following two design requirements:

- The client shall take a message on the command line; this message is passed to the server's "hello" operation.
- The client shall expect the server's stringified object reference in the file "server.ior" in the current directory.

First, the client loads the Combat ORB.

```
package require combat
```

Then, the ORB is initialized. The application's command line is passed to the ORB, so that the user may specify ORB options on the command line. Non-ORB options are returned from the corba::init command and again captured in the argv variable.

```
set argv [eval corba::init $argv]
```

The client expects a message on the command line, and complains if the command line is empty.

```
if {[llength $argv] == 0}
  puts stderr "usage: $argv0 <message>"
  exit 1
}
set message $argv
```

Now the client loads type information for the "HelloWorld" interface into Combat's type repository. This is done by simply reading the file "hello.tcl" that was generated by the idl2tcl application above, using the "source" command.

```
source hello.tcl
```

The client next reads the stringified object reference from the "server.ior" file that the client is expecting in the current directory. One option is to use the "open" and "read" commands; another is to use a "file"-style stringified object reference. These require an absolute path name, which "pwd" provides.

```
set obj [corba::string_to_object file://[pwd]/server.ior]
```

The "obj" variable now holds an object reference that points to the server, and the client is ready to make remote invocations on the server according to the "HelloWorld" interface. The "hello" method takes a single string parameter; the client is passing the message from the command line.

```
$obj hello $message
```

The client can also read the server's "messageCounter" attribute:

```
set messageCounter [$obj messageCounter]
puts "The server's message counter is $messageCounter."
```

Done with the server, the client can release the object reference and exit.

```
corba::release $obj
```

7.3 The Server

The server is also implemented as a command-line application. For simplicity, there are only a few design requirements:

- The server writes its stringified object reference to the file "server.ior" in the current directory.
- The server runs perpetually (until it is manually terminated, e.g., by typing Ctrl-C). The server just runs in the foreground, i.e., it is not implemented as a "daemon."

First, the server program loads the Combat ORB.

```
package require combat
```

Then comes the definition of the "HelloWorld_impl" class, which implements the "HelloWorld" interface: it implements the "hello" method and has the "messageCounter" attribute. As explained in section 4.1, the class inherits from the base class "PortableServer::ServantBase" and also implements the "_Interface" method which informs Combat of the IDL type that is being implemented.

```
itcl::class HelloWorld_impl {
  inherit PortableServer::ServantBase
  public method _Interface {} {
    return "::HelloWorld"
  }
  public variable messageCounter 0
  public method hello {message} {
    puts "The client says: $message"
    incr messageCounter
  }
}
```

Like in the client, the server initializes the ORB, passing any command-line parameters to the ORB, and loads type information for the "HelloWorld" interface into Combat's type repository.

```
eval corba::init $argv
source hello.tcl
```

Servants must be activated with an instance of a Portable Object Adapter (POA). The Root POA local object is acquired using corba::resolve_initial_references. The program also queries the POA Manager instance, which is later needed to switch the POA to the "active" state.

```
set poa [corba::resolve_initial_references RootPOA]
set mgr [$poa the_POAManager]
```

Now the server can create an instance of the HelloWorld_impl class, like any other [incr Tcl] object. For simplicity, the instance is activated with the Root POA itself rather than creating a new POA instance – for the limited purposes of this example, the Root POA's policies are acceptable.

```
set srv [HelloWorld_impl #auto]
set oid [$poa activate_object $srv]
```

Next the server wants to write the servant's stringified object reference to the console. This is done by first calling the POA's "id_to_reference" method, which returns an object reference, and then the ORB's `corba::object_to_string` method to turn the object reference into a stringified object reference.

```
set ref [$poa id_to_reference $oid]
set ior [corba::object_to_string $ref]
puts "$ior"
```

The same string can then be written to the "server.ior" file in the current directory.

```
set iorfile [open "server.ior" w]
puts -nonewline $iorfile $ior
close $iorfile
```

Finally, the server activates the POA – this is done via its manager – and enters the event loop.

```
$mgr activate
vwait forever
```

Because the variable "forever" is never written to, the "vwait" command never returns but keeps running Tcl's the event loop, in which the ORB listens for incoming requests.

7.4 Running The Example

Running the example is trivially done by opening two consoles, and changing to the "demo/hello" directory. In the first console, run the server:

```
./server.tcl
```

The server should print its stringified object reference (a long hexadecimal string) to the console, and the file "server.ior" should appear. Then, run the client, passing it a message on the command line:

```
./client.tcl "It worked."
```

This message should then appear in the server's console.

8 Issues And Workarounds

8.1 Object References Without Type Information

As explained elsewhere, Combat normally uses type information from its Interface Repository to properly format CORBA messages. This relies on the ability to extract the *Repository Id* from object references – this Repository Id is then used as an index into the Interface Repository, i.e., the data that was loaded from idl2tcl-generated files.

In some cases, however, the object reference may not contain the desired Repository Id:

- "corbaloc:" stringified object references contain addressing but no type information.
- Even in "IOR:" stringified object references, the Repository Id field is optional and may be absent. (Admittedly, I have not seen this occur "in the wild.")
- Sometimes a stringified object reference contains the Repository Id for a base interface rather than the desired most-derived interface.

The symptom that results from using an object reference without type information is that all remote method invocations will fail with the "IDL:omg.org/CORBA/INTF_REPOS:1.0" exception.

The easiest workaround is to confirm the object's type with the built-in "_is_a" operation. Given a Repository Id or a scoped name as a parameter, it queries the remote server whether the servant supports this type, returning true or false. If the result is true, Combat takes notice and remembers that the object reference supports this type; future remote method invocations will succeed. Consider this example of accessing a CORBA Naming Service:

```
% source CosNaming.tcl ;# generated by idl2tcl
% set o [corba::string_to_object corbaloc::localhost:1627/NameService]
_combat_obj_1
% $o to_name "a/b"
IDL:omg.org/CORBA/INTF_REPOS:1.0 {minor 0 completion_status COMPLETED_NO}
% $o _is_a ::CosNaming::NamingContextExt
1
% $o to_name "a/b"
{id a kind {}} {id b kind {}}
```

Note how the `corba::string_to_object` operation returns a valid object reference, but, even though we have loaded type information for the Naming Service into the Interface Repository, using the object reference fails at first because the "corbaloc:" string does not contain the Repository Id. Then we use the "_is_a" operation to validate that this object is indeed of the expected type. Combat then remembers that this object reference implements this type, and from then on, invocations succeed.

A few more notes:

- This is issue mostly occurs at bootstrapping only. Object references that are returned from a method invocation (e.g., in this document's initial example, where the Bank's "create" operation returned an object of type "Account") typically contain proper type information.
- Before giving up, Combat attempts to query the remote object for its type information using the CORBA Reflection specification. If all servers implemented this specification, there would be no need to feed Combat's Interface Repository. Unfortunately, this specification is not widely supported yet.
- Combat also asks the remote object for its own CORBA Interface Repository (using the built-in "_interface" operation). If servers were configured to use an Interface Repository, there would be no need to feed Combat's own. However, it is relatively uncommon to set up Interface Repositories. If you want to go this route, consult your ORB documentation.
- Combat can also access CORBA Interface Repositories instead of its own, if one is configured using the "-ORBInitRef InterfaceRepository" option at ORB initialization time.
- You can use the "iordump" tool to check if a stringified object reference contains correct type information.

8.2 Missing Type Information

Note that the "IDL:omg.org/CORBA/INTF_REPOS:1.0" exception is also raised if type information is not available in Combat's Interface Repository, e.g., because the application neglected to load the type information, regardless of whether the object reference contained a Repository Id or not. In this case, even the "_is_a" workaround above will not help.

8.3 Proprietary Bootstrapping

Some ORBs implement proprietary bootstrapping functionality, i.e., means for a client to discover servers without the need to transport an object reference. E.g., the TAO ORB implements a multicast protocol that allows clients to discover the Naming Service. However, Combat does not implement these ORB-specific protocols. Therefore, even though your C++ clients may not need an explicit reference to the Naming Service or other services, Combat clients do.

8.4 Non-Functional DNS

Combat by default uses the local hostname, as returned from Tcl's "info hostname" command, as addressing information in the object references that it generates. Clients must therefore be able to resolve this hostname into an IP address. If that is not the case, you can use the "-ORBHostName" option at ORB initialization time to set a host name or IP address that Combat will use instead of "info hostname:"

- In a local network that does not implement DNS, you may want to use your local IP address.
- When using Combat over the internet, you may want to use a fully qualified host name.

8.5 Incompatible Codesets

Connections may rarely fail because of incompatible character sets ("codesets"). When a client opens a CORBA connection to a server, it sends its codeset information. If the server does not know about the client's codeset, it may decide to abort the connection.

Combat by default uses the codeset returned by Tcl's "encoding system" command. If that codeset turns out to be incompatible, you can use the "-ORBNativeCodeSet" option at ORB initialization time to use a different codeset, which is specified as a Tcl encoding name (i.e., it must match one of the strings returned from Tcl's "encoding names" command).

Combat uses a small built-in table to map Tcl's encoding names to values from the "OSF Character and Code Set Registry." This table is at the top of Combat's "codeset.tcl" file. If ORB initialization fails with the "unknown native codeset" error, it may be necessary to extend this table.

8.6 Interface Repository Discrepancy

There is the potential for discrepancy between Combat's Interface Repository if the original IDL file (and the server implementation) has been modified but the idl2tcl-generated file has not been updated. This can result in subtle issues ranging from immediate connection failures to misinterpretation of data.

Note that this situation is not unique to Combat; it is equivalent to independently compiling and linking client and server programs using incompatible IDL.

8.7 Reentrancy

Because the ORB enters Tcl's event loop when waiting for replies to any outstanding requests, there is the possibility that other events are processed. E.g., Tk might react to GUI events. That potentially allows for reentrancy, e.g., if a button is pressed while a remote invocation is being made in reaction to a button being pressed. An application that processes events from multiple sources needs to take appropriate precautions so that this situation does not cause any issue, e.g., by ignoring GUI events while remote invocations are being performed. Applications can generally be written to tolerate reentrancy.

8.8 Leaking Memory

Just to reiterate: because object references are associated with individual Tcl commands, they consume memory that is not reclaimed when the variable that contains the object reference's name goes out of scope. All object references must eventually be passed to the `corba::release` command to avoid leaking memory. While it may be acceptable to be sloppy about memory leaks in small scripts, it may eventually exhaust available memory in scripts that run for a long time and that "go through" many object references. Either way, it is good practice to be aware of existing object references and to free unused object references.

This applies to:

- Object references returned from `corba::string_to_object` operation.
- Object references returned from the POA, e.g., from the "id_to_reference" operation.
- Object references returned from method invocations, including object references returned in "inout" or "out" parameters.
- Object references that are returned as members of a complex data structure.

9 To Do

Combat is reasonably complete. Some random leftover food for thoughts:

- Multithreading is not yet supported, at least not explicitly. It is perfectly possible to load the ORB into multiple independent Tcl threads, but that results in multiple independent ORBs. It would be nice if there was just one thread-aware ORB.
- There is no collocation optimization. I.e., when a client calls a server that is implemented in the same process, the request and reply are transported over TCP/IP.
- [incr Tcl] is nice and allows the server-side language mapping to be very similar to C++, but it does not support virtual inheritance.
- It would be nice to avoid idl2tcl's dependence on another ORB. It would be even cooler if Combat could directly read IDL files. However, parsing IDL is frustratingly complex (mostly due to the IDL language's "#pragma prefix" kludge), and then you'd still be dependent on an external C preprocessor.

Note that the author's motivation for future development is partly fueled by user feedback. I would love to hear of projects using Combat, or of plans to use it.

10 History

Combat first saw light in 1998 as "TclMico," written in C++, using the Dynamic Invocation Interface. It was tightly coupled with the MICO ORB, using some of its proprietary APIs to map between Tcl's and MICO's type system event loop. For a while, Combat lived on the "bleeding edge," with frequent updates as Tcl, MICO, the CORBA specification and compilers matured. Even the Unix landscape was much more diverse then, with considerable acrobatics required to support multiple platforms.

TclMico was renamed to "Combat" with version 0.6 in 2000 when the implementation was ported to other ORBs. The advent of the Portable Object Adapter and Dynamic Any specifications largely abolished the reliance on ORB-specific extensions. As a somewhat undesired side effect, this turned Combat into an excellent test suite for these parts of the CORBA specification – with a considerable number of bug reports against pretty much every ORB that Combat was tested against. At first, ORBacus was the only ORB beside MICO that qualified.

The name "Combat" was chosen out of belief that a catchy, snappy name is better than an accurate but boring one. It may be expanded as "Corba Object Management By Application of Tcl" if you like, but who cares? However, "Combat" has proven a poor choice because it is "ungoogleable." Searching for "TclMico" proved very accurate, whereas searching for "Combat" ... you can imagine the result. Even with more qualifications to the search you get mostly false positives.

Building Combat remained a challenge. A diverse landscape of ORBs, operating systems, compilers, and their versions made the entire process somewhat hit and miss. E.g., on some platforms, it was not straightforward to load C++ shared libraries into a C executable like Tcl because of a static "libstdc++." Some platforms still shipped without ISO C++ support, and gcc availability was far from universal. It was frustrating to spend so much effort on porting.

Having been involved with the development and therefore very familar with the internals of the MICO ORB, I eventually began to apply that experience to the implementation of a stand-alone ORB entirely written in pure Tcl, avoiding any ORB or platform compatibility issues. Initially released in January 2001 with limited client-side only functionality, the ORB rapidly matured.

Combat version 0.7 was released in October 2001 as both "Combat/C++" and "Combat/Tcl," with near equivalent functionality.

Combat 0.7 received a few patches over the following year but proved very robust for a wide range of applications. One of the more significant additions came in 2004, when Combat joined MICO and Orbix as the first ORBs to implement the CORBA Reflection specification.

Between 2004 and 2008, a few minor bug fixes trickled into Combat/Tcl, but Combat/C++ saw little use. The pure-Tcl ORB proved just too convenient – it required no porting effort at all between platforms. While Combat/C++ may be faster, Combat/Tcl's performance proved more than sufficient for my occasional use of Combat for ad-hoc GUIs. Therefore I decided to retire Combat/C++. Version 0.8 of the Tcl ORB took over the name "Combat" without the "/Tcl" qualifier.

The original intention was to submit Combat to the Object Management Group as an official Tcl language mapping for CORBA, and to reserve the "1.0" version number until that time. However, with Tcl's popularity eclipsed by other scripting languages over the years, that never came to pass.