Rapid CORBA Server Development in Tcl: A Case Study

Jason Brazile and Andrej Vckovski
Netcetera AG
{jason.brazile,andrej.vckovski}@netcetera.ch

Abstract

A large Swiss bank needed to collect, combine, process, and distribute financial market data from various 3rd party data sources to a large number of internal and external clients – the typical integration task at which scripting languages excel. The bank uses an implementation of CORBA as their standard enterprise-wide middleware for distributed applications. We describe how we designed and built a Tcl/C++ transport framework which allowed us to develop the "kernel" of this server application entirely in Tcl, yet support CORBA as the primary interface to the server. We further describe how this framework allows a small development team to rapidly implement changes and enhancements to the server and its external interface, while automatically generating the corresponding changes that are needed for the CORBA interface. Additionally, we show how we were able to automatically generate code to create new tcl commands that make use of the same, generated, Tcl/C++ marshalling routines in order to develop a CORBA client in Tcl, which is used to regression test the server, when full end-to-end testing is needed.

1 Overview

In his keynote address at the 1999 USENIX Technical Conference [1], John Ousterhout argued that typical programming projects are shifting away from large stand-alone applications and moving toward *integration* applications, (sometimes also referred to as *mega-programming* [2]). The importance of these applications, he argued, lies not in providing fundamental new features as much as in the ability to coordinate and extend existing applications – exactly the tasks at which scripting languages excel. He argued that many things taken for granted today, such as strong typing and inheritance, may not make sense in most future applications.

As we analyzed our customers requirements for for their strategic new mission critical server application, we found ourselves in agreement with this basic philosophy. It became clear that our customer needed to primarily integrate information coming in from several already existing applications – combining, merging, and formatting everything into a unified view. This result would need to be exported through certain transport mechanisms – initially the most important being CORBA. However, even at the earliest stages, other access channels (e.g. XML) were planned.

Upon noting CORBA as a requirement, many software architects might automatically dismiss the idea of designing the core application functionality in a scripting language such as Tcl, even though it is not without precedent [3], [4]. However, we were further encouraged to follow our core-as-scripting-language idea when we realized that our application needed to be prepared for the following:

- Frequent changes to the interfaces to external applications
- Rapid integration of new data sources (i.e., new applications)
- Frequent changes to the access requirements requested by clients (i.e., IDL changes)
- Rapid implementation of new features requested by marketing analysts
- Possible implementation of different data access channels (e.g. XML) as some move out of favor, and others move in

It is worth mentioning that this server application is mission critical in the sense that some client applications planned to present the data on the Internet (i.e., they are highly visible) for tasks such as financial planning, online banking, and as a financial news source. An even

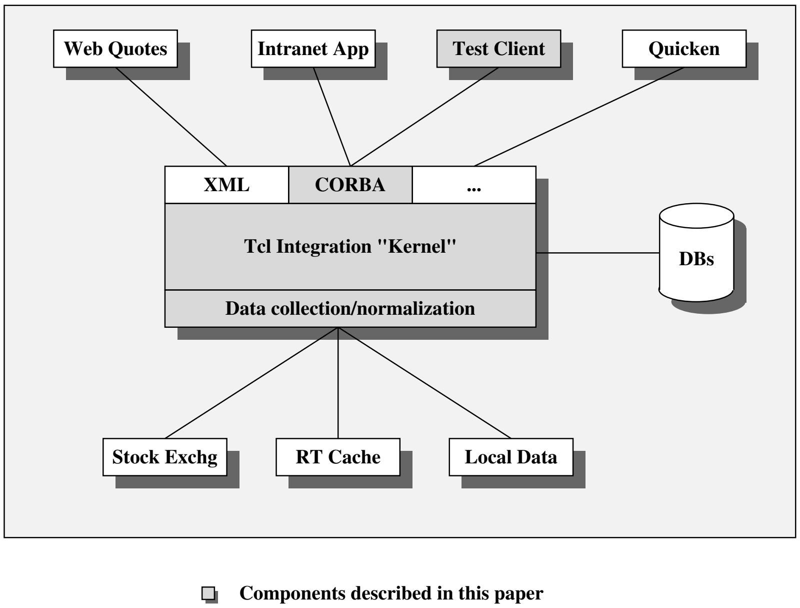

Figure 1: Simplified Server Overview

greater sense of urgency was placed upon us when we learned that some of the client applications were planned to go online within weeks of our planned initial release – many of these applications with budgets 10 times larger than ours.

However, these robustness and flexibility requirements merely further increased our resolve to attempt to develop as much as possible in a scripting language and as little as possible in traditional CORBA server implementation languages.

Our goal then became to design and implement a flexible platform that performed these integration tasks – providing data in a canonical form and otherwise acting as mediator to a large and growing set of heterogeneous data sources with different manners of access to the same interfaces.

2 Architecture

The overall feature that we were targetting was a one-to-one mapping between the methods defined by the CORBA IDL (Interface Definition Language) specification and the Tcl procedures that would implement these specifications. To illustrate this with an example, consider the following IDL definition for a method called `foo::square()`:

```
interface foo {
  int square (
    in  int a,
    out int b
    );
}
```

We would like to define a Tcl procedure to implement this method that looks like this:

```
proc foo::square {a vb} {
   upvar $vb b
   set b [expr $a * $a]
}
```

The general idea would be that a CORBA request coming in to the server would get translated to a corresponding Tcl command. Then a Tcl interpreter would be invoked that first “sourced” the Tcl implementations of these commands, and then “eval’d” the Tcl command that was composed. It would then translate the Tcl results of that evaluation back into CORBA objects which are returned to the client.

With this design, the framework would then consist of a CORBA server with an embedded Tcl interpreter that would source the “real methods” implemented as procedures in a Tcl script “kernel”.

However, in addition, we wanted to extend a Tcl shell that would serve as a CORBA client which would con-

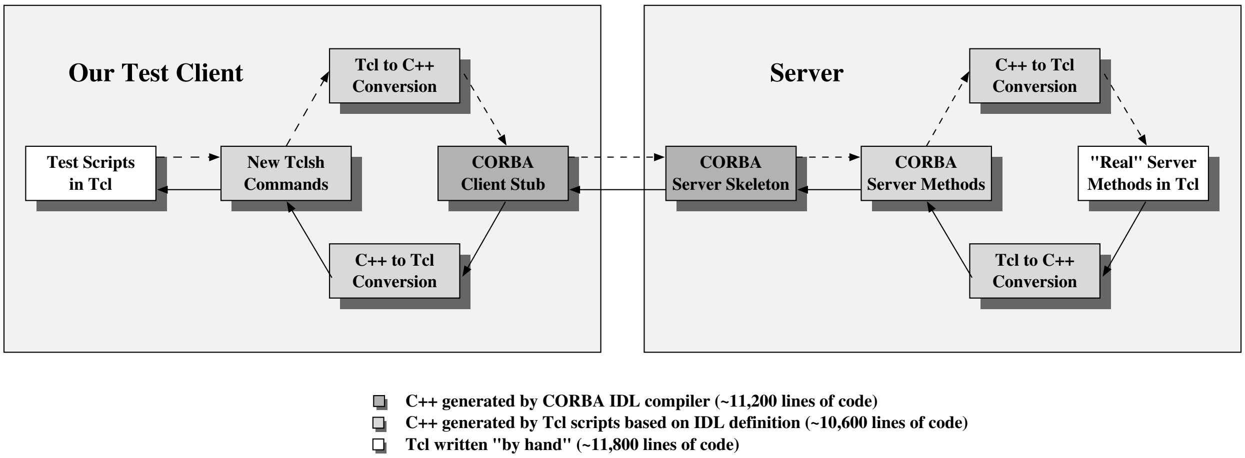

Figure 2: Data Flow (CORBA Case)

tain Tcl commands that also had a one-to-one mapping to the methods defined in the IDL. The function signatures of these new commands would look identical to the function signatures in the procedures in the Tcl “kernel”. This would allow for natural looking Tcl code as in the following example:

```
% foo::square 4 result
% set result
16
```

One of the best characteristics of this architecture is that test scripts could be written in Tcl that could be run either on the CORBA Tcl client or in a “shortcut” path which completely bypasses any middleware infrastructure by making direct access to the Tcl “kernel” procedures. This feature allowed much of the server functionality to be able to be developed on a stand-alone UNIX laptop which did not have a CORBA development environment.

To summarize, this design approach would give us the following benefits:

- The ability to begin development of the core methods in Tcl completely independent of, and in parallel to, the development of CORBA support.
- The ability to write test scripts in Tcl
- The ability to develop all the core programming logic in a rapid turnaround scripting language, rather than a lower level, strictly typed, compiled language.
- If different transport paths are required, only a new transport layer would need to be written and the same “kernel” could be used to give identical functionality with maximal code reuse.

After implementing a small prototype to test the feasibility of this approach, it became clear that a large amount of the CORBA transport layer code was regular enough that it could be automatically generated – again by scripts written in Tcl.

The full data path and amount of automatic code generation we ended up with is shown in Figure 1.

At the end of the design phase, we concluded that the key aspects of this architecture required the following components to be written:

- CORBA Language mappings for Tcl
- Methods/Operations implemented in Tcl
- Automatic generation of the marshalling code based on IDL definitions
- Automatic generation of client side tcl commands based on IDL definitions

3 Language mappings

One of the key issues to resolve was how to map CORBA objects to typeless Tcl strings. In the worst case, this could be done by defining every C++ value to be a Tcl list containing two elements – the value and its type. Then, a lookup table could be constructed to store the types and whatever needed semantics might be associated with it in order to manipulate or access objects of that type.

Fortunately, however, a much simpler strategy was possible in this case. We ended up using a mapping similar to that used by Pilhofer in Tclmico [3]. We started by looking at all of the possible CORBA types that are made available – the basic data types like boolean, integer, and floats, as well as the compound data types such as structures, arrays, unions, and sequences. We also wanted to be able to map Tcl exceptions to CORBA exceptions.

It is fairly straightforward to map the common types such as a structure. For example, an object of type `RequestContext` which can be defined like this:

```
typedef sequence <string> Profile;
struct rc {
  int     sessionId,
  string  application,
  string  lang,
  Profile profile,
  string  user,
};
typedef rc RequestContext;
```

could become a Tcl list with the following representation:

```
set my_rc {-1 web-quotes ENGLISH \
   {SWISS_REALTIME US_DELAYED} guest}
```

Going from this Tcl list representation back to its corresponding C++ object representation requires that we know the CORBA types of each component. However, this is positionally implicit based on the context of the procedure calls that use variables of these types and the definitions of the procedures themselves as defined in the IDL. In other words, when we know that something is the first argument to method `getMarketData` and according to our codification of the IDL, the first argument to method `getMarketData` is of type `RequestContext`, then we just pass this list to a routine that converts a list (which in this case, itself contains a list) to a C++ object of type `RequestContext`. This can be done statically, because we always know at compile time which procedures are being called and what types their arguments are because of their position. That is, except in the case of translating "exceptions", which is described below.

To take a quick glance at some of the other types, a variable whose type is a `union` can be mapped to a two-element list where the first element is the discriminator and the second is the corresponding value. The `sequence` and `enum` types are simply mapped to a list.

Perhaps the trickiest mapping to come up with was a mapping for exceptions. A Tcl exception (as thrown by the `error` command) allows only one string/list as an argument. However, for our purposes, we needed to know both *what* exception occurred (i.e., an exception type) and an additional exception-specific object. The problem of course is that this second object's "shape" can be different depending on what type of exception is to be thrown.

One interesting point to be made is that by having a general mapping mechanism for Tcl exceptions, our application can catch not only application specific exceptions that we throw ourselves, but also the standard Tcl exceptions that may occur due to attempted undefined variable access, for example.

We imposed a convention upon our own code whereby our application specific exception objects would have a certain well-defined structure, so that we could then differentiate them from Tcl exceptions which get mapped to an "Internal Error" exception with a Tcl backtrace, rather than merely crashing the application.

This turned out to be the one case where we needed to determine type information dynamically in order to be able to do the CORBA language mappings. The code is structured as follows:

```
if (Tcl_EvalObj() == TCL_ERROR) {
  if (/* has special shape */) {
    type = Tcl_GetStringFromObj();
    ex_type = type_to_enum(type);
    switch (ex_type) {
    case exceptionA:
      // now we know the type
      Convert::from_tcl(...);
      THROW(exceptionA, ...);
    case exceptionB:
      // now we know the type
      Convert::from_tcl(...);
      THROW(exceptionB, ...);
    }
  } else {
    // Tcl Exception
    THROW(internal, ...);
```

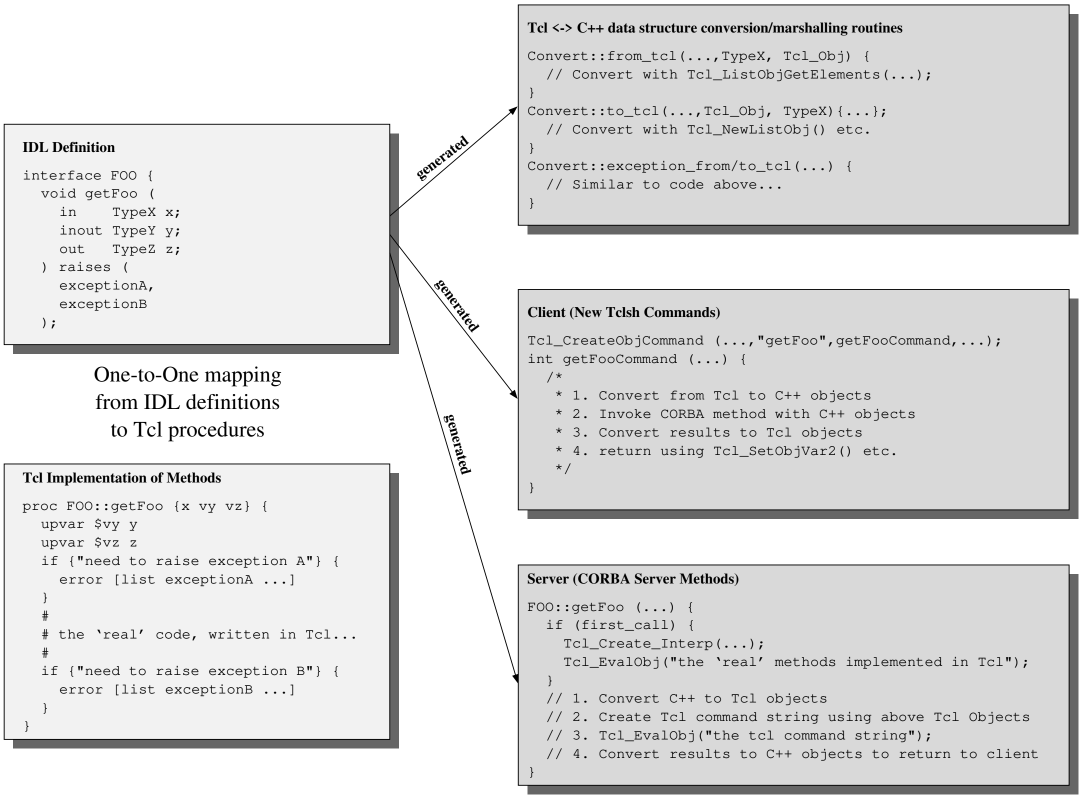

Figure 3: Automatic Generation of CORBA “Transport” Routines

```
    }
}
```

Raising an exception from within our Tcl code looks like this:

```
if {$some_exceptional_condition} {
  error {badParam \
    {431 "FOO parameter expected"}}
}
```

4 Code Generation

One of the most important features of the framework is that a large portion of the low-level (e.g. non-Tcl) code can be automatically generated, (Figure 2). This is good not only for quick response to changes in requirements, but also in our level of confidence in the server’s robustness. Once we developed a logical, regular pattern for use in code generation, and we measure and test that code for relative correctness and memory leaks etc., then we have a high degree of trust in code that is generated from the same patterns in future enhancements. And as a practical matter, we feel confident that if the generated code checks return values and correctly handles error conditions in one case, it will also correctly handle them in all similar cases.

To recap, our design revealed that the following components could be generated automatically:

- Marshalling code from/to Tcl
- Implementation of new Tcl commands for a Tcl interpreter used as a CORBA client
- CORBA server skeletons which translate a method to a Tcl command, “eval” the Tcl “kernel”, and translate the result back.

We used our hand-written prototype implementation as a model for the generated C/C++ code. There is nothing particularly noteworthy about the Tcl scripts which generated the C/C++ code. It was mostly a matter of template textual substitution.

It should be pointed out that the code generation scripts are not necessarily general purpose. Code was only written to generate the types and cases that appeared in our server IDL definition. As new methods and data types were added to the IDL definition, additional lines of code generation code occasionally had to be written. On the other hand, none of these additions so far have caused any fundamental change in the general code generation technique that is used.

It should further be pointed out that the code is not generated *directly* from the IDL definition files, but rather from a "pre-parsed" intermediate representation of the information that is represented in the IDL definition files.

The IONA Orbix CORBA product provides a code generation toolkit [5] which exposes a pre-parsed representation of IDL files through Tcl data structures and accessor procedures by way of a built-in Tcl interpreter. This would be an efficacious approach for our application, but at the time of implementation, we were limited to a proprietary CORBA environment which therefore required our own ad hoc solution.

Our simplistic intermediate file representation was designed so that it can be used in a Tcl code generation script by merely "eval'ing" it. We wanted to defer the difficult problem of parsing the IDL files themselves in the hopes that an automatic solution, such as that provided by Orbix, would eventually become available. In the meantime, making these changes to our intermediate file "by hand", whenever the IDL file has changed, has not required a significant effort.

4.1 Marshalling

The marshalling code was built around the idea of having two essential polymorphic conversion methods – `Convert::from_tcl()` which would take a Tcl object representing a list structure as input and produce a CORBA object (which may be composed of CORBA sub-objects) as output – and `Convert::to_tcl()` which would go the opposite direction. The types of the arguments to these routines would determine which instance of the conversion routine would be called.

It was also decided that these routines would recursively call themselves to process each subcomponent that might need to be converted, thus eliminating code duplication.

A representative pair of automatically generated conversion methods is shown for our `RequestContext` object in Figure 3. Notice that the fourth member of the structure is a non-primitive object which leads to a recursive call to `Convert::from_tcl`.

4.2 Server

On the server side, the IDL compiler generates server skeletons (i.e. procedure stubs) for the methods that are defined in the IDL. However, our Tcl script subsumes this behavior by generating its own completed server skeletons which merely rely on there being a correspondingly named Tcl procedure in the Tcl "kernel" that it can "eval".

An example of generated server code is shown in Figure 4.

It might be worth mentioning that the Tcl "kernel" is packaged as a monolithic static string in a shared object so that it appears as a standard looking shared library. This is advantageous not only for simpler code distribution, but has a higher management acceptance factor than scripts as text files.

4.3 Client

On the client side, much of the code is analogous to what is required for the server. The arguments need to be converted, the relevant method in the new implementation language needs to be called, then the output from the result needs to be translated back.

An example of generated client code is shown in Figure 5.

5 Testing

By far, the greatest advantage in testing was the ability to write tests in Tcl and to merely source our "kernel" implementations of the methods being tested. A quick change could be made and the method in question could be re-"sourced" with rapid turnaround.

Also, with this method, the important code could be written without the need for a CORBA development in-

```
int Convert::from_tcl(Tcl_Interp *interp,
      RequestContext *&requestContext,
      Tcl_Obj *tcl_obj)
{
  Tcl_ListObjGetElements(interp, tcl_obj, &obj);
  /* verify that there are 5 arguments */
  Tcl_GetLongFromObj(..., obj[0], &(requestContext->sessionId));
  requestContext->application = Tcl_GetStringFromObj(..., obj[1], ...);
  requestContext->lang = Tcl_GetStringFromObj(..., obj[2], ...);
  Convert::from_tcl(interp, &(requestContext->profile), obj[3]);
  requestContext->user = Tcl_GetStringFromObj(..., obj[4],...);
}

int Convert::to_tcl(Tcl_Interp *interp,
      Tcl_Obj **tcl_obj,
      RequestContext *requestContext)
{
  *tcl_obj = Tcl_NewListObj(0, NULL);
  Tcl_ListObjAppendElement(Tcl_NewLongObj(requestContext->sessionId,...));
  Tcl_ListObjAppendElement(Tcl_NewStringObj(requestContext->application,...));
  Tcl_ListObjAppendElement(Tcl_NewStringObj(requestContext->lang,...));
  Convert::to_tcl(interp, &tcl_profile, &(requestContext->profile));
  Tcl_ListObjAppendElement(Tcl_NewStringObj(requestContext->user,...));
}
```

Figure 4: Simplified Data Marshalling code (CORBA Case)

```
void MDS::getMarketData (
  /* in */  const RequestContext &requestContext,
  /* in */  const DataSelector   &dataSelector,
  /* in */  const ReturnFields   &returnFields,
  /* out */ const MarketData     &results,
)
{
  interp = Tcl_CreateInterp();
  Tcl_EvalObj(/* the "kernel" */);
  Convert::to_tcl(interp, &tcl_request_context_obj, &requestContext);
  Convert::to_tcl(interp, &tcl_data_selector_obj, &dataSelector);
  Convert::to_tcl(interp, &tcl_return_fields_obj, &returnFields);
  tcl_command_obj = Tcl_NewListObj(0, NULL);
  Tcl_ListObjAppendElement(interp, tcl_command_obj, /* "getMarketData" */);
  Tcl_ListObjAppendElement(interp, tcl_command_obj, tcl_request_context_obj);
  Tcl_ListObjAppendElement(interp, tcl_command_obj, tcl_data_selector_obj);
  Tcl_ListObjAppendElement(interp, tcl_command_obj, tcl_return_fields_obj);
  Tcl_EvalObj(interp, tcl_command_obj);
  tcl_market_data_results_obj =
    TclObjGetVar2(interp, /* "results" */, ...);
  Convert::from_tcl(interp, &results, &tcl_marketdata_results);
}
```

Figure 5: Simplified Generated Server code (CORBA Case)

```
void getMarketDataObjCmd(Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[])
{
  orb_init(&corba_obj);
  Convert::from_tcl(interp, &objv[0], &requestContext);
  Convert::from_tcl(interp, &objv[1], &dataSelector);
  Convert::from_tcl(interp, &objv[2], &returnFields);
  TRY (
    corba_obj->getMarketData(requestContext, dataSelector,
      returnFields, marketDataResults);
  )
  CATCH (InternalError) { /* Handle Exception */ }
  CATCH (badRequestContext) {/* Handle Exception */ }
  Convert::to_tcl(interp, &tcl_marketdata_results, MarketdataResults);
  Tcl_ObjSetVar2(interp, /* "marketDataResults" */, tcl_marketdata_results);
}
```

Figure 6: Simplified Generated Client code (CORBA Case)

frastructure or even the need to run separate server/client processes.

Once we had a CORBA Tcl client, we were able to reuse the same testing scripts we wrote during development to do end-to-end testing across the CORBA channel. This even allowed other development groups who were writing CORBA-based client applications to our server, to use our CORBA Tcl client in order to do quick exploration and cross-checking of the interface whenever they ran into difficulties.

An example test script looks like this:

```
% set requestContext {$sessionID \
    $applicationID ENGLISH \
    {DELAYED} $userId}
% set dataSelector {BYCONSTRAINTS \
    {{} {"US Dollars"} \
    {"Swiss Exchange"}}}
% set returnFields   {BYFIELDNAME \
    {ID CURRENT_PRICE HIGH_PRICE}}
% getMarketData $requestContext \
    $dataSelector $returnFields \
    results
% puts $results
{ID 324598234 CURRENT_PRICE 45.5
HI_PRICE 48.25} {ID 43098234 ...}
```

6 Conclusion

In summary, with our CORBA-as-transport design, we were able to achieve:

- Rapid development
- Robustness
- Ease of testing
- Rapid implementation of interface changes
- Development outside of huge support environment
- Reuse of useful tools in the OSF arena to integrate other applications and leverage all of this even within a rigidly specified middleware framework.

At least up to a certain point, performance was never a high-priority requirement. However, it turned out not to be a problem either. When performance problems were encountered, the largest gains were achieved by adding caching at the Tcl "kernel" level to compensate for slow data accesses to external applications.

The most unexpected benefit was the ease in providing an XML access method to our Tcl "kernel" which obviously gives identical semantics, results, and performance characteristics. We were able to use many of the same code generation techniques to write automatic XML to Tcl marshalling/unmarshalling code, which relies on exactly the same pre-parsed intermediate representation of the IDL that the CORBA transport layers uses.

We must admit that there was initial skepticism in the approach we took, especially at the beginning of the project when so much effort seemed to be needed in just building framework and code generation tools which didn't lead directly to our tightly scheduled goal.

However, in the end we not only achieved our goal but were easily able to adapt to even more change requests than anticipated, given the high level of flexibility our system afforded. The IDL specification has gone through 12 revisions since the application was first launched in December 1998.

7 References

1. Ousterhout, John, *Integration Applications: The Next Frontier in Programming*, Keynote Address, 1999 USENIX Technical Conference, Monterey, California, 1999.

2. Wiederhold, Gio, Peter Wegner, and Stefano Ceri, *Towards Megaprogramming*, Communications of the ACM, Vol.,35 No.11, November 1992.

3. Miller, Michael and Kareti, Srikumar, *Using Tcl to Script CORBA Interactions in a Distributed System*, The Sixth Annual Tcl/Tk Conference, San Diego, California, 1998.

4. Pilhofer, Frank, *Tclmico – A Tcl interface to the Mico ORB* `<http://www.vsb.informatik.uni-frankfurt.de/~mico>`

5. *Orbix Code Generation Toolkit Programmer's Guide*, IONA Technologies PLC, Dublin, Ireland, February 1999.