Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (0223) 323010**
INTERNATIONAL: **+44 223 323010**
FAX: **+44 223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Prototyping tools for dependability

Owen Rees

Abstract

Mechanisms to support dependable distributed systems can be expensive both to implement and to use. Research results suggest that properties of applications can be used to deploy cost-effective dependability mechanisms, but these results need to be verified. Designs derived from the research results need to be presented in a form that can be used by programmers implementing industrial quality dependability infrastructures.

Exploration of the research results requires a rapid prototyping environment in which the various options can be tested without undue effort. It is also important to be able to explore what is happening in the system being prototyped. The prototyping language needs a combination of power and simplicity so that the important design issues are not obscured by a mass of detail.

This document describes prototyping tools intended to satisfy these objectives, and their application to the development of a dependability infrastructure and programming model. The tools provide interactive object oriented distributed processing, with high level graphical user interface support.

This work is being carried out as part of ANSA Phase III task D2.

APM.1238.00.01 **Draft** 7 July 1994

Request for Comments (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

Prototyping tools for dependability

Prototyping tools for dependability

Owen Rees

APM.1238.00.01

7 July 1994

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(0223) 323010
INTERNATIONAL	+44 223 323010
FAX	+44 223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Overview

1.1 Objectives

1.1.1 Exploration of dependability mechanisms

The dependability infrastructure components, and the underlying prototyping technology, were developed in order to explore the problem of making dependable distributed systems more cost-effective.

1.1.2 Transfer of results to ANSA sponsors

The underlying prototyping technology is based on software that is freely available with no usage or copying restrictions. The underlying software has been ported to a wide variety of platforms.

ANSA sponsors will be able to use the prototyping technology to explore the results in detail, and to adapt the results to their own requirements.

1.2 Structure of the document

The prototyping tools consist of an interpreter, and a collection of libraries.

The interpreter supports the Tool Command Language (Tcl) [OUSTERHOUT 94] with extensions to support object-oriented programming, distributed processing, and a high level graphical user interface. The interpreter, and the components from which it is built, are described in chapter 5 with some simple examples to illustrate the various parts.

A basic interaction library packages the facilities provided by the interpreter so as to provide transparent remote object invocation, this is described in chapter 2. The interaction library is intended to be easy to adapt to the needs of the prototyping work.

Some basic services are described in chapter 3. These are services that will be used in the experiments on dependability; they also illustrate the use of the interaction library.

An additional library adds support for replication that can be transparent to the application level objects; this library is described in chapter 4. The library implements particular policies, and the descriptions include some discussion of the policy choices, and the impact of alternatives upon the implementation.

The order of presentation of these topics is intended to reflect the motivation for developing the prototyping environment, and to illustrate how ic an be used.

2 Interaction support

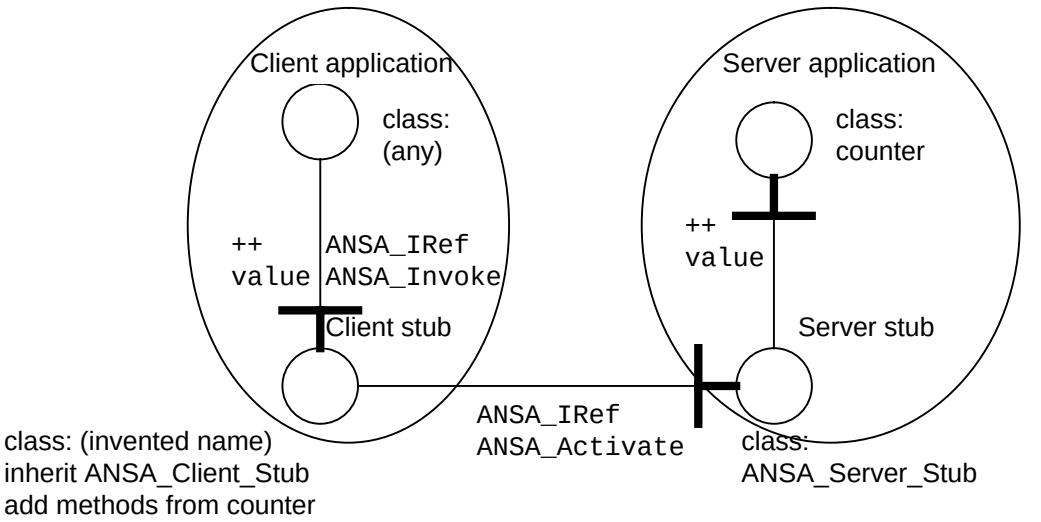
2.1 Overview

This chapter describes the mechanisms that provide support for interaction between [incr Tcl] objects located in different processes, which may be on different hosts. The primary objective was to provide transparency; the nature of that transpaceny is one of the issues discussed below.

2.2 Basic mechanism

Figure 2.1 illustrates the basic structure of a server that has made an interface available.[1] The example is a trivial counter with methods '++' and 'value', like the one described in §3.5.

Figure 2.1: Client and server stub objects



Within the server there is a counter which may be invoked by a name local to that server. Transmitting the use of that counter outside the server causes the infrastructure to be put in place for the client to invoke the counter as if it were local.

2.2.1 Marshalling for transmission

In the prototype, most of the work is done as part of the marshalling of the interface for transmission.

1. Note that this interface may have been passed as an argument or result of any invocation.

```
# N.B. In the stub array, client stubs point to themselves, local
# instances point to their server stubs. The ANSA_IRef method
# delivers the ref for the server stub in both cases. If there is
# no stub, make a stub for local instances, and return other
# things unchanged

proc ANSA_marshall {item {name \#auto}} {
  global ANSA_Host ANSA_reminsts ANSA_Stubs
  if {[info exists ANSA_Stubs($item)]} {
    return [$ANSA_Stubs($item) ANSA_IRef]
  }
  if {[ANSA_Local_Instance $item]} {
    [ANSA_Server_Stub $name $item] ANSA_IRef
  } {
    return $item
  }
}
```

The prototype exploits the associative arrays of Tcl in order to keep track of all the stubs that have been created. If a stub for the counter already exists, it will be reused. The first time the interface is marshalled, there is no server stub so one must be created.

Once the stub has been found, its method `ANSA_IRef` is invoked to deliver the string to be transmitted as the reference. Note that the “on the wire” format for the reference is hidden in the server stub, so different reference formats can be accommodated by different stub classes.

The function ANSA_Local_Instance is used to distingush between instances of objects and other entities. Since Tcl deals only with strings, local instances are passed by a string that names them. It is assumed that any string that is the name of a local instance is to be treated as a reference to the instance, and this is the test performed by the function.

```
proc ANSA_Local_Instance {instance} {
  string compare "" [itcl_info objects $instance]
}
```

2.2.2 Creating a server stub

The dynamic evaluation mechanism of Tcl makes it possible to have a generic server stub class. It will be invoked by a client stub with the name of the required method as an argument.

```
itcl_class ANSA_Server_Stub {
  constructor {i} {
    global ANSA_Host ANSA_Stubs ANSA_reminsts
    set inst $i
    set iclass [$i info class]
    set myref "iref://$ANSA_Host:[ANSA_Server]/$iclass/$this"
    set ANSA_Stubs($i) $this
    set ANSA_reminsts($myref) $i
  }

  method ANSA_IRef { } { return $myref }

  method ANSA_Activate {method arglist} {
```

```
      set largs {}
      foreach arg $arglist { lappend largs [ANSA_unmarshall $arg] }
      set result {}
      foreach r [eval $inst $method $largs] {
        lappend result [ANSA_marshall $r]
      }
      return $result
    }
    protected inst
    protected myref
  }
```

An instance of this class is created by the call:

```
ANSA_Server_Stub $name $item
```

The stub has a name by which it can be invoked. This is allocated automatically by giving #auto as the name argument, except for certain bootstrapping cases. The class constructor function is called with the rest of the arguments; in this case, the invocation name of the local instance for which this is to be the stub.

The constructor captures the argument in an instance variable and creates the string that will be transmitted as the reference. The stucture is modelled upon URLs as used in the world wide web. The generated string will be something like:

```
iref://plato:1554/counter/ANSA_Server_Stub0
```

This includes:

1. the host name
2. the port on which this interpreter is listening for incoming invocations
3. the name of the class of the object for which this is the stub
4. the invocation name of this server stub

Note that the port is relative to the host, and the names of the class and the stub are relative to the host:port pair. The way this information is used is described in §2.2.4 *Unmarshalling an incoming reference.*

The constructor also makes entries in two arrays.

The first array – `ANSA_Stubs` – remembers the mapping from the local object instance to the stub so that the same stub can be used again next time the interface is marshalled.

The second array – `ANSA_reminsts` – is used to avoid unnecessary creation of client stubs. In this case, if the reference is ever unmarshalled locally, the local instance can be used directly so it is entered as the instance to use when unmarshalling the reference.

Note: Are these merely optimisations, or does everything break if the arrays are not used? This needs to be checked.

2.2.3 Becoming a server

The first time a server stub is created, the interpreter must establish a port listening for incoming requests. In the prototype, the same port is used for all incoming requests.

```
set ANSA_Host [exec hostname]

proc ANSA_Server {{port 0}} {
  global ANSA_Server_port
  if {![info exists ANSA_Server_port]} {
    dp_Host +*.ansa.co.uk
    set ANSA_Server_port [dp_MakeRPCServer $port dp_CheckHost]
  }
    return $ANSA_Server_port
}
```

The host name is remembered in a global variable set when the interaction library is loaded.

The port number is remembered in a global variable that is set the first time the port function is called. The function is called with no argument and allocates an arbitrary port number, except for certain bootstrap cases. Note that some simple access control is applied to the port when it is opened.

2.2.4 Unmarshalling an incoming reference

The client stub is created by the unmarshalling of an incoming reference.

```
# Called with a potential iref, converts it if it is one,
# otherwise leaves alone

proc ANSA_unmarshall {iref} {
  global ANSA_remclasses
  global ANSA_reminsts

  if {[info exists ANSA_reminsts($iref)]} {
    return $ANSA_reminsts($iref)
  }
  set re {^iref://([^:]*):([^/]*)/(.*)/(.*)$}
  if {[regexp $re $iref x host port rclass inst]} {
    set ci "$host:$port:$rclass"

    if {![info exists ANSA_remclasses($ci)]} {
      set allmethods [ANSA_Call $host $port $inst info method]
      set usermethods [ANSA_methods $allmethods]
      set cname [ANSA_GenClass ANSA_Client_Stub $usermethods]
      set ANSA_remclasses($ci) $cname
    }
    $ANSA_remclasses($ci) \#auto $iref $host $port $inst
  } {
    return $iref
  }
}
```

If a stub already exists for this reference, it is re-used. Otherwise the reference is unpicked with a regular expression to extract the information necessary to create the client stub.

The stub must be an instance of a class that supports the operations in the class for which it is a proxy. Client stubs cannot be completely generic, but can inherit common behaviour and instance variables from a base class. In order to create the stub, the referenced instance is invoked to discover what methods it supports:

```
set allmethods [ANSA_Call $host $port $inst info method]
```

This invocation reports all the methods, including various built-in methods, such as the destructor. It also reports fully-qualified names[1] in the context of the referenced instance. Since the built-in methods will need to operate on the client stub object, rather than the referenced instance, the list of method names must be processed to remove built-in names and strip off the class prefixes.

```
set usermethods [ANSA_methods $allmethods]
```

The need for this filtering is a consequence of using the built-in description facility "`info method`", rather than constructing a description mechanism do do this specific job. The filter function is straightforward, it goes through the list picking out the names of the methods that are neither built-in not part of the interaction infrastructure.

```
# return a list of the interesting methods in some list
proc ANSA_methods {rawlist} {
  set methods {}
  foreach cm $rawlist {
    if {[regexp {(ANSA)?.*::(.*)$} $cm x ansa method]} {
      if {[string compare $ansa ""]==0} {
        switch $method {
          config { }
          constructor { }
          delete { }
          isa { }
          default { lappend methods $method }
        }
      }
    }
  }
  return $methods
}
```

Once the list of methods has been constructed, the required class can be constructed – the function `ANSA_GenClass` described in §2.2.5 does this.

```
set cname [ANSA_GenClass ANSA_Client_Stub $usermethods]
```

Another array – `ANSA_remclasses` – is used to keep track of which remote classes have local proxy classes already. This avoids the potentially remote interation to discover the method names.

```
set ANSA_remclasses($ci) $cname
```

Once the class has been looked up or created, a new instance of that class can be created as the local proxy for the referenced instance:

```
$ANSA_remclasses($ci) \#auto $iref $host $port $inst
```

The effect of this instantiation is determined by the class constructor which is described in §2.2.6.

1. I.e. names with a class prefix such as `counter::++` rather than just `++`.

In this version of the prototype, only the names of the methods are retrieved. One of the objectives of the work is to explore the use of additional information about the methods – in particular their use of mutable state. The simple lookup and local filtering will need to be replaced by a more detailed interface description retrieval in order to support that work.

2.2.5 Generating a new class

The prototype infrastructure exploits the interpreter's ability to create new classes dynamically. A function is provided to generate a new class with a set of parents and a set of methods passed as parameters.

```
proc ANSA_GenClass {parents methods} {
  global ANSA_Classes ANSA_remclasses
  set key [list $parents [lsort $methods]]
  if {![info exists ANSA_Classes($key)]} {
    if {![info exists ANSA_remclasses(next)]} {
      set ANSA_remclasses(next) 0
    }
    set ANSA_Classes($key) GENCLASS[incr ANSA_remclasses(next)]I

    set classdef "itcl_class $ANSA_Classes($key) {"
    if {[string compare $parents ""]!=0} {
      append classdef " inherit $parents;"
    }

    foreach m $methods {
      append classdef "method $m {args}"
      append classdef " { eval ANSA_Invoke $m \$args  }; "
    }
    append classdef "}"
    eval $classdef
  }
  return $ANSA_Classes($key)
}
```

This function constructs an `itcl_class` command, with an inherit clause and method bodies derived from the arguments. This command is then evaluated to create the class.

Classes that have already been constructed are remembered in the array `ANSA_Classes` which is indexed by the parent class names and method names. The order of parent classes may be significant, but the order of methods is not; this is reflected in the key used as the array index.

Note that all the method bodies have the same structure:

```
method name {args} { eval ANSA_Invoke name $args };
```

The special argument name `args` is used to gather all the arguments into a list and `eval` is used to pass the list elements as individual arguments to the method `ANSA_Invoke`[1]. The method `ANSA_Invoke` is expected to be inherited from a parent class, so different effects are achieved by inheriting from different parents.

1. Since the `ANSA_Invoke` method also gathers its arguments into a list, the use of eval could be avoided by a consistent change to the code generated by `ANSA_GenClass` and the `ANSA_Invoke` methods of the base classes.

Additional information about the methods would permit the generated methods to have the correct number of arguments, and the use of eval could be avoided.[1]

2.2.6 The client stub base class

The ANSA_unmarshall function specifies the class ANSA_Client_Stub as the parent class for the classes it generates.

```
itcl_class ANSA_Client_Stub {
  constructor {iref rhost rport rif} {
    global ANSA_Stubs ANSA_reminsts
    set rref $iref
    set host $rhost
    set port $rport
    set rinst $rif
    set ANSA_Stubs($this) $this
    set ANSA_reminsts($iref) $this
  }

  method ANSA_IRef { } { return $rref }

  method ANSA_Invoke {method args} {
    eval ANSA_Call $host $port $rinst $method $args
  }

  protected rref
  protected host
  protected port
  protected rinst
}
```

The constructor function saves the arguments in instance variables for future use. The constructor also updates the arrays ANSA_Stubs and ANSA_reminsts so that this stub will be invoked to provide the reference string when marshalling, and will be re-used when that reference string is unmarshalled.

This class provides two methods that will be inherited by its children.

ANSA_IRef is used in marshalling, and delivers the reference for the service for which this stub is a proxy. Note that server stubs have the same method, so the marshalling function need not distinguish between client stubs that are proxys for remote services, and server stubs for local services.

ANSA_Invoke is the method invoked from the generated method bodies described in §2.2.5. This method calls ANSA_Call (see §2.2.7) which makes a remote call, passing the saved data about the service for which this stub is a proxy.

2.2.7 Making remote calls

The function ANSA_Call makes a remote call and waits for the result.

1. If the correct number of arguments were generated, either the ANSA_Invoke method would have to gather the arguments into a list, or the generated methods would have to combine the separate arguments into a single argument.

```
proc ANSA_Call {host port instance method args} {
  set largs {}
  foreach arg $args { lappend largs [ANSA_marshall $arg] }
  set peer [dp_MakeRPCClient $host $port]
  set robj [ANSA_Promise \#auto $peer]
  dp_RDO $peer ANSA_Request $robj $instance $method $largs
  return [$robj Collect]
}
```

It first marshalls the arguments:

```
set largs {}
foreach arg $args { lappend largs [ANSA_marshall $arg] }
```

then establishes a connection to the server using the function `dp_MakeRPCClient` provided by the Tcl-DP package:

```
set peer [dp_MakeRPCClient $host $port]
```

The call itself is made using the non-blocking call mechanism, passing the local name of a “promise” object to be used to collect the response.

```
set robj [ANSA_Promise \#auto $peer]
dp_RDO $peer ANSA_Request $robj $instance $method $largs
```

finally, the result is collected from the promise, and returned.

```
return [$robj Collect]
```

The use of non-blocking calls and promises is not necessary in this case, but is used because it is necessary to support active replication. This is discussed in chapter 4.

2.2.8 Receiving requests

The function `ANSA_Request` is called at the server when a request arrives.

```
proc ANSA_Request {robj instance method arglist} {
  global rpcFile errorInfo
  set rfile $rpcFile
  if {[catch {$instance ANSA_Activate $method $arglist} r]} {
    dp_RDO $rfile $robj Error $r $errorInfo
  } {
    dp_RDO $rfile $robj Response $r
  }
}
```

This function invokes the `ANSA_Activate` method of the server stub named in the request, catching any error that may occur. It then invokes the response object (the promise at the client end) to deliver the result or error code. The response is delivered using the connection that was established by the client; this connection remains open throughout the call.

The `ANSA_Activate` method of the server stub (see §2.2.2) performs all of the unmarshalling of received arguments, and marshalling of results to be returned. Note that this means that error results and associated information will not be marshalled, but simply passed as a string.

2.2.9 Receiving responses

Responses are delivered by invoking a method of a promise instance.

```
itcl_class ANSA_Promise {
  constructor {peer} {
    global ANSA_Promise
    set file $peer
    set ANSA_Promise($this) 1
  }
```

The constructor saves the file handle for the connection to the server, and sets an element of a global array to indicate that the promise is not yet ready to be redeemed. A global array element is used since the wait mechanism used by the `Collect` method described below requires a global.

```
  method Response {result} {
    global ANSA_Promise
    set r {}
    foreach res $result { lappend r [ANSA_unmarshall $res] }
    dp_CloseRPC $file
    set ANSA_Promise($this) 0
  }
```

The `Response` method is called by the server to deliver a normal result. It unmarshalls the results, closes the connection to the server, and sets the global aray element to indicate that the result is available.

```
  method Error {result errorInfo} {
    global ANSA_Promise
    set r $result
    set errInfo $errorInfo
    set failed 1
    dp_CloseRPC $file
    set ANSA_Promise($this) 0
  }
```

The `Error` method is called by the server to deliver an error result. It saves the error information, closes the connection to the server, and sets the global aray element.

```
  method Collect { } {
    global ANSA_Promise
    while {$ANSA_Promise($this)} {
      dp_waitvariable ANSA_Promise($this)
    }
    if {$failed} { error $r $errInfo } { return $r }
  }
```

The `Collect` method is called to collect the result. It will wait until the result is available, then either return the result or signal the error.

```
  protected r
  protected errInfo
  protected failed 0
  protected file
}
```

These are the instance variables of a promise.

2.3 Design issues

2.3.1 Choices

Some aspects of the design are driven by the needs of the replication mechanisms. In particular, the use of non-blocking calls rather than the blocking RPC provided by Tcl-DP is a consequence of the replication strategy.

The placement of the marshalling and unmarshalling is more a consequence of the evolution of the prototype than any explicit design choice.

2.3.2 Exploiting features of the interpreter

Some aspects of the design exploit features of the interpreter, and would need to be reconsidered in other environments.

The objects are self-describing, and this is exploited in order to retrieve the name of the class of the object, and the names of the methods of an object.

The ability to create classes dynamically is exploited to construct fully transparent proxies for arbitrary classes. Being an interpreter that can be used interactively, it is necessary to construct proxies that offer the full service. Where the client has a more static definition, it would be more usual to construct the proxy that will satisfy the client and then bind it to the server provided that a conformance test is passed.

3 Basic services

3.1 Overview

This chapter describes the basic services that have been implemented using the interaction library, in order to support the experiments. These fall into two categories: supporting services and example services.

Supporting services are simplified forms of services that are likely to exist in many real systems. A trading service is the only example available at present. The primary purpose of the trading service in the prototype is to provide a fixed point form which to start.

Example services are services constructed purely to aid in the experiments. A simple counter is the example service used throughout this document.

Both kinds of service use a common management facility to do simple houskeeping.

3.2 Managed services

The primary function of the management facility is to keep track of exports and withdraw the offers when the service shuts down.

```
itcl_class management {
  constructor {{s "Server"}} { set idstring $s }

  method shutdown { } {
    puts "$idstring shutting down"
    foreach offer $exports {
      catch {$offer withdraw}
    }
    destroy .
    exit
  }

  method export {ctxt inst} {
    set type [$inst info class]
    set props [list "pid [pid]"]
    lappend exports [$ctxt export $inst $type $props]
  }

  protected exports {}
  protected idstring
}
```

The `export` method supplies the type and properties for the export, and remembers the offer interface. The `shutdown` method uses the remembered offer interfaces to withdraw the offers.

3.3 The trading service

The prototype trading service is organised as context objects that provide searching over offer objects. In the current implementation, only one context is used. The trader also has a graphical interface that displays the offers, and has buttons that allow the offers to be checked for validity, and a button to shut down the trader.

3.3.1 Startup and management

As mentioned above, the prototype trader supports one context and displays the state of that context. It is implemented as a script so that it can be run from a shell prompt.

```
#!/usr/local/bin/idpwish -f

cwdlib

wm title . "Trivial Trader"
```

The first line identifies this as a script to be run by the idpwish interpreter (§5.3.2). The `cwdlib` function causes the current directory to be used as a library of Tcl functions[1] – this is used as an aid to development. For the benefit of the experimenter, the window title is set to “Trivial Trader”.

```
set w {}
set base [context \#auto [ListBox $w.base]]
```

This creates a context object, passing it a `ListBox` in which to display itself. The `ListBox` class is supplied with [incr Tcl]; it displays a list of items, with an automatically managed scrollbar.

```
if {[catch {ANSA_Server 2094}]} {
  puts "Failed to run trader ([pid]) shutting down"
  destroy .
  exit
} {
  set title "[ANSA_marshall $base base] ([pid])"
  puts "Base context is at $title"
}
```

This is where the trader establishes itself in a well known place. The `ANSA_Server` function is called with an argument that specifies the port to be used. The marshalling function is then called with the extra argument that specifies a name for the server stub. The information that might be needed by the experimenter is written out; for example, if the script is in a file called trader.tcl it couuld be run like this (input in bold):

```
121% ./trader.tcl &
[3] 13719
122% Base context is at iref://plato:2094/context/base (13719)
```

This information includes the string that can be unmarshalled into a proxy for the context that provides the trading service interface.

1. Tcl can load a library script the first time a function it contains is used. [incr Tcl] extends this to allow loading of classes on first use.

```
management manage "Trader at $title"
```

A management object named ‘manage’ is then created for the interpreter.

```
label $w.title -text "Trivial Trader: $title" -relief ridge
pack $w.title
```

The reference string and process id are displayed in the trader’s window. The final form of the window on startup is shown in figure 3.1.

Figure 3.1: The trader graphical interface on startup


```
pack $w.base -expand 1 -fill x -anchor n
```

The ListBox created earlier is packed below the title line; it is initially empty and so appears as a blank space.

```
pack [set w [frame $w.buttons]] -fill x

button $w.check -text Check -command "$base check"
pack $w.check -side left

button $w.quit -text Shutdown -command "manage shutdown"
pack $w.quit -side right
```

Finally, the rest of the graphical interface is created. The buttons are set up to invoke the appropriate method of one of the objects just created.

3.3.2 The context class

The trading service interface is provided by a context object.

```
itcl_class context {
  constructor {lb} {
    set offers(x) array
    unset offers(x)
    set lbox $lb
    $lbox config -list {}
  }
```

The constructor forces the `offers` 'variable' to be an empty array, and remembers the associated list box which is configured to be empty.

```
  method export {inst type props} {
    set o [offer \#auto $this $inst $type $props]
    set offers($o) 1
    set item [list [list $o [$inst ANSA_IRef] $type $props]]
    $lbox config -list [concat [$lbox get] $item ]
    return $o
  }
```

The export method is provided for use by servers. For servers that use the management facility, it is invoked from within the management object as shown in §3.2.

An offer object is created, and added to the offers array. Remembering offers by using them as array indexes simplifies the removal of an offer from the context.

A string describing the offer is constructed and added to the list box. The string contains information that may be useful to the experimenter.

```
  method import {type constraint} {
    foreach o [array names offers] {
      if {[$o isOfType $type] && [$o matches $constraint]} {
        return [$o instance]
      }
    }
    return {}
  }
```

The import method is provided for use by prospective clients that are looking for a service. This implementation uses a simple "first match" strategy, and returns the selected instance if there is one. The type and constraint tests are performed by methods of the offer, so it is the offer class that determines the nature of the tests that are performed.

```
  method lookup {type constraint} {
    set result {}
    foreach o [array names offers] {
      if {[$o isOfType $type] && [$o matches $constraint]} {
        lappend result $o
      }
    }
    return $result
  }
```

The lookup method returns all of the offers that match the type and constraints. Note that it is the offers that are returned, rather than the instances to which they refer. This means that the prospective client may invoke other methods supported by the offer.

```
    method redisplay { } {
      set l {}
      foreach o [array names offers] {
        set ref [[$o instance] ANSA_IRef]
        lappend l [list $o $ref [$o type] [$o properties]]
      }
      $lbox config -list $l
    }
```

The redisplay method is used internally to bring the list box up to date.

```
    method withdraw {o} {
      set r [expr 1-[catch {unset offers($o)}]]
      redisplay
      return $r
    }
```

The withdraw method removes an offer from the context. This method is invoked from within the offer in order to remove the offer from the context. It is not intended to be used remotely.

There are some naming issues which can be explored here. The offer must know the context that contains it, and the name by which it is known in that context. That name need not be known or meaningful elsewhere.

```
    method check { } {
      foreach o [array names offers] {
        set i [$o instance]
        if {[$i isa ANSA_Client_Stub]} {
          lappend i ANSA_Invoke
        }
        if {[catch {eval $i info class}]} {
          unset offers($o)
        }
      }
      redisplay
      return {}
    }

    protected offers
    protected lbox
  }
```

The check method removes offers for which the instances no longer exist. It does this by invoking the `info` method which all [incr Tcl] objects support. Since this includes the local client stubs for remote services, it is necessary to detect the local client stubs, and use the stub method `ANSA_Invoke` explicitly.

3.3.3 The offer class

```
itcl_class offer {
  constructor {ctxt inst type props} {
    set p(x) make-it-an-array
    unset p(x)
    set i $inst
    set t $type
    set context $ctxt
    foreach prop $props {
      set p([lindex $prop 0]) [lindex $prop 1]
    }
  }
```

The offer constructor remembers the parameters passed to it, turning the properties which were passsed as a list of lists into an array.

```
  method isOfType {type} {
    if {[string compare $type $t]==0} { return 1 }
    if {[string compare $type any]==0} { return 1 }
    return 0
  }
```

The `isOfType` method in the prototype implements a simple exact match of names policy. In order to be able to retrieve all of the offers in a context, a special type name any has been added; every type is considered to be of this type.

```
  method matches {constraint} {
    foreach c $constraint {
      set pname [lindex $c 0]
      if {![info exists p($pname)]} { return 0 }
      if {[string compare [lindex $c 1] $p($pname)]} {
        return 0
      }
    }
    return 1
  }
```

The prototype implements a very simple propery matching policy. The constraint is a list of {name value} lists; each property named in the constraint must exist with exactly the specified value.

```
  method instance { } {set i}
  method type { } {set t}
  method properties { } {
    set r {}
    foreach pn [array names p] {lappend r [list $pn $p($pn)]}
    return $r
  }
```

These three methods can be used to retrieve the information about the offer. Note that the properties are returned as a list of lists, in an order determined by Tcl's `array names` command.

```
  method withdraw { } {
    $context withdraw $this
  }

  protected context
  protected i
  protected t
  protected p
}
```

Finally, the `withdraw` method which invokes the `withdraw` method of the context, and the instance variables.

3.4 The trader client library

A library is provided to make it easy for applications to use a trading service running on the same host. This strategy is intended to avoid conflicts between experimenters, but makes it harder to conduct experiments that are distributed across several hosts. The client library can easily be changed to use a specific host or to use information in an environment variable. The appropriate balance between ease of cooperation and avoidance of unwanted interaction will depend upon the task in hand.

The library must establish a client stub for the trader; the unmarshalling function is used to do most of the work.

```
proc trader {args} {
  global trader
  if {![info exists trader]} {
    set ir iref://[exec hostname]:2094/context/base
    if {[catch {ANSA_unmarshall $ir} trader]} {
      error "Cannot contact trader"
    }
  }
  eval $trader $args
}
```

If the global variable trader has not yet been set, an attempt is made to unmarshall the reference to the local trader. This will fail if the trading service is not running at the expected location; the unmarshalling function invokes the trading service in order to construct the correct client stub class, as described in §2.2.4.

Finally, the trading service is invoked through the interaction mechanisms described in chapter 2. Here is an example of importing and using the counter described in §3.5 (input in bold).

```
% set c [trader import counter {}]
GENCLASS2I0
% $c ++
1
% $c ++
2
%
```

3.5 The counter service

The counter service is an example used to conduct experiments. It is a very simple service with mutable state and a graphical display that shows the current value, and also allows the counter to be shut down.

```
#!/usr/local/bin/idpwish -f
cwdlib
itcl_class counter {
  constructor {w} {
    set disp $w
    redisplay
  }

  method value { } { return $count }
  method ++ { } { incr count; redisplay; value }
  method redisplay { } { $disp configure -text $count }

  protected count 0
  protected disp
}
```

The counter, like the trader, is implemented as a script file. The counter class defines a counter that displays its value.

```
if {$tcl_interactive==0} {
  management manage "Counter ([pid])"

  wm title . "Counter"
  set w {}
  pack [label $w.title -text "Counter: pid [pid]" -relief ridge]
  pack [label $w.value -relief sunken]
  pack [button $w.quit -text Shutdown -command "manage shutdown"]

  manage export trader [counter \#auto $w.value]
}
```

The mangement and display are not set up if the file is loaded into an interactive interpreter. This condition was introduced to simplify some transparency experiments.

Setting up the management and the display are similar to the trader (§3.3.1). The counter exports its interface through the management service so that it will be withdrawn when the counter is shut down. Running the counter script creates a counter with an initial display as shown in figure 3.2.

Figure 3.2: Initial display for a counter

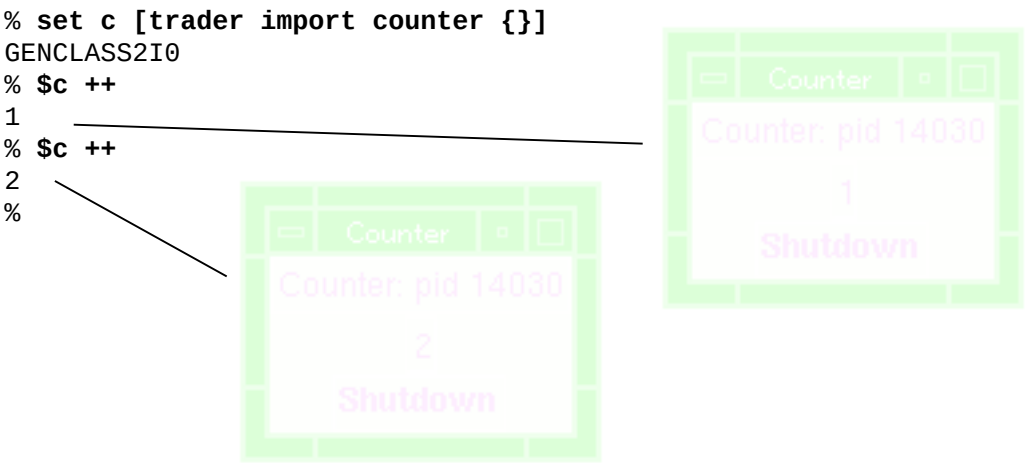
```
130% ./counter.tcl &
[4] 14030
131%
```

The trader now has an offer, so its display is updated as shown in figure 3.3

Figure 3.3: Trader display after counter export

A client can import and then invoke the counter as was described in §3.4. The invocations of the counter increment operation cause the display to be updated as illustrated in figure 3.4

Figure 3.4: Counter display updates



The ability to observe the counter values changing is useful when experimenting with replication. This is described in chapter 4.

4 Replication support

4.1 Overview

The objective of this part of the work is to provide a kit of parts that can be combined to explore replication issues. In particular, the kit of parts is to be used to explore the effect of various quorum, collation, and ordering policies, and to demonstrate the relationship between feasible policies and propagation of knowledge between client and server.

The replication support library is adapted from the interaction support library described in chapter 2, and uses the classes and functions defined in that library.

This chapter describes the curent state of the replication support library, a number of the elements required for replication have not yet been implemented.

4.2 Group prototype

In order to support replication, the marshalling and unmarshalling functions were changed to use different classes when instantiating server stubs and when generating classes for client stubs. This means that clients and servers are always prepared to interact with and to be part of a group. Interacting between groups, and clients or servers that do not have built-in group support, requires further study.

4.2.1 Group server stubs

Like the basic server stub, the group server stub is generic

```
itcl_class ANSA_Group_Server_Stub {
  inherit ANSA_Server_Stub
```

The group server stub inherits from the basic server stub; this includes the mechanisms to invoke the local instance, and managing the stub, reference and local instance mappings.

```
  constructor {i} {
    ANSA_Server_Stub::constructor $i
    set members $myref
  }
```

The constructor function initialises the list of members to be this stub only. This must be done after the basic stub constructor fills in the instance variables; the default constructor call order must be overridden.

```
  method checkInc {ci} {expr $ci==$incarn}
```

The checkInc method is used to check that an incarnation identifier is current. This is used to check incarnation identifiers sent by clients.

```
method updateClient {rfile cstub} {
  dp_RDO $rfile $cstub ANSA_Membership $incarn $members
}
```

The updateClient method invokes a method in the client stub to update its view of the group incarnation and membership.

```
method membership {incid newmembers} {
  set incarn $incid
  set members $newmembers
  return
}
```

The membership method updates the stub's incarnation identifier and list of members. A check that the stub is in the list of members could be inserted here.

```
  protected members
  protected incarn 1
}
```

The instance variables for a group server stub are the list of members which is initialised by the constructor, and the incarnation id which is initilaised to 1.

4.2.2 Receiving requests at a group

Note: This is one of the functions that needs to be replaced in order to support replication properly.

This is the function called at the server when the client believes that it is interacting with a group. The client passes extra information compared with a call of the basic ANSA_Request function (§2.2.8).

```
proc ANSA_MRequest {ro cstub sinc actid ccard inst meth argl} {
  global rpcFile errorInfo
  set rfile $rpcFile
  if {
      [catch {
        if {[$inst isa ANSA_Group_Server_Stub]} {
          if {![$inst checkInc $sinc]} {
            $inst updateClient $rfile $cstub
          }
        }
        $inst ANSA_Activate $meth $argl
      } r]
    } {
    dp_RDO $rfile $ro Error $r $errorInfo
  } {
    dp_RDO $rfile $ro Response $r
  }
}
```

If the invoked instance has a group stub, then the incarnation identifier sent by the client is checked by calling the stub method checkInc described in §4.2.1.

If the client's view is out of date, this implementation adopts the policy of attempting to update the client's view of the membership, and going ahead with the invocation. This would cause problems in the absence of a consistency protocol within the group[1].

The quorum and order processing for server consistency has not yet been implemented, nor has the quorum and collation processing for invocations from groups.

4.2.3 Group Client Stubs

Like the basic client stub, the group client stub is used as a base class when creating specialised client stubs.

```
itcl_class ANSA_Group_Client_Stub {
  inherit ANSA_Client_Stub
```

The group client stub inherits from the basic client stub; this provides the stub, instance and reference management

Note: Inheriting from the basic stub is of doubtful value given the nature of groups. The `check` method of trading `context` objects uses an `isa` test for client stubs. Removing the inheritance link might create problems since group client stubs would no longer be considered to be client stubs. This is an example of the problems that arise when a test of implementation strategy is used where information about purpose and meaning is required.

```
constructor {iref rhost rport rif} {
  ANSA_Client_Stub::constructor $iref $rhost $rport $rif
  unset members
  set members($iref) 1
}
```

The constructor initialises the membership list. In this case it is kept as a array, permitting some information about each member to be kept. This facility is not exploited at present.

The constructor assumes that the group has one member, this assumption is implicit in the unmarshalling function and the basic client stub.

```
method ANSA_Invoke {method args} {
  set refs [array names members]
  ANSA_Collate [ANSA_MCall $refs $this $servinc $method $args]
}
```

The `ANSA_Invoke` method overrides the inherited version. It invokes a multiple call function, and collates the results which are returned as a list. This client stub and its view of the server incarnation identifier are passed as parameters.

1. The consistency protocol has not yet been implemented, so operations that update group state, invoked by clients with an obsolete view, cause the state to become insonsistent.

```
    method ANSA_Membership {incarnation newmembers} {
      set servinc $incarnation
      unset members
      foreach member $newmembers { set members($member) 1 }
      return
    }
```

The method ANSA_Membership is called to set the incarnation and membership of the group. This method is called by the group server stub in the current implementation.

```
    protected servinc 1
    protected members {}
  }
```

Always creating stubs for groups with one member, and fixing them up when the first invocation occurs, is a simple strategy, but with some problems. If the member fails or leaves the group before the first invocation, then the client has lost contact with the group, despite its continued existence.

The advantage of this approach is that the references passed in invocations are simple, they do not need to be able to accommodate groups.

The underlying problem is of how groups are named in references passed as arguments, and what those names really mean. Singling out groups as a special case allows various implementation options to be used, but leaves open the question of why groups deserve special treatment.

A very simple collation function is used in the prototype, it requires that all the results are identical (as strings - the only Tcl type.)

```
proc ANSA_Collate {results} {
  set r1 [lindex $results 0]
  foreach rn [lrange $results 1 e] {
    if {[string compare $r1 $rn]!=0} {
      error "Collation failed"
    }
  }
  return $r1
}
```

4.2.4 Making a multiple call

The group client stubs use a multiple call function.

```
proc ANSA_MCall {irefs cstub sinc method arglist} {
  global ANSA_Promise
  set largs {}
  foreach arg $arglist { lappend largs [ANSA_marshall $arg] }
  set waitlist {}
  set actid 1
  set ccard 1
  set $re {^iref://([^:]*):([^/]*)/(.*)/(.*)$}
  foreach iref $irefs {
    if {[regexp $re $iref x host port rclass inst]} {
      set peer [dp_MakeRPCClient $host $port]
      set robj [ANSA_Promise \#auto $peer]
```

```
        lappend waitlist $robj
        dp_RDO $peer ANSA_MRequest $robj $cstub $sinc \
                     $actid $ccard $inst $method $largs
      }
    }
    set results {}
    foreach robj $waitlist {
      lappend results [$robj Collect]
    }
    return $results
  }
```

This function is similar to the basic call function; the difference is that it makes several non-blocking calls, with a promise for each, and then collects all of the results.

The use of non-blocking call is essential here; all the calls must be in progress concurrently since the members will need to negotiate a processing order before processing any of the calls. Tcl does not have any direct support for concurrency so the only option left is to use a non-blocking call.

4.2.5 Managing server stubs

The functions described so far do not provide any way to invoke the server stub methods that deal with membership. A support fuction has been provided in order to do this.

```
  proc ANSA_MStubOp {irefs stubop args} {
    set results {}
    set re {^iref://([^:]*):([^/]*)/(.*)/(.*)$}
    foreach iref $irefs {
      if {[regexp $re $iref x host port rclass inst]} {
        set peer [dp_MakeRPCClient $host $port]
        lappend results [eval dp_RPC $peer $inst $stubop $args]
        dp_CloseRPC $peer
      }
    }
    return $results
  }
```

This differs from the multiple call function in that it uses the blocking call, and invokes a method named in a parameter on the server stub instances.

Until proper group membership management has been implemented, this function can be used directly to set up membership lists in the server stubs. A group membership manager would use this, or a derived function to perform its function.

4.2.6 Using the replication library

The basic services described in chapter 3 can be used to demonstrate the replication library functions in action. The simplest demonstration is to create two counters, join them into a group, and then update both with a single invocation from the client.

The situation after creating a trader and two counters is shown in figure 4.1. At this stage, the counters are not related, each can be invoked separately. The counters can be distinguished by their process IDs which are shown in the display and are also properties of the offer..

Figure 4.1: Two separate counters

```
140% ./trader.tcl &
[3] 19924
141% Base context is at iref://plato:2094/context/base
(19924)
./counter.tcl &
[4] 19928
142% ./counter.tcl &
[5] 19933
143%
```


The various services can be invoked from an interactive interpreter:

```
143% idpwish
% cwdlib
0
% set c1 [trader import counter {{pid 19928}}]
GENCLASS2I0
% set c2 [trader import counter {{pid 19933}}]
GENCLASS2I1
% $c1 ++
1
% $c2 ++
1
%
```

The counters each display the value 1 after these invocations.

The counters can be joined into a group; the lack of state synchronisation means that this must be done while their states are consistent, as they are now.

```
% set xx {}
% foreach offer [trader lookup counter {}] {
    lappend xx [ANSA_marshall [$offer instance]]
}
% ANSA_MStubOp $xx membership 2 $xx
{} {}
%
```

This joins all of the counters (§3.5) that have exported their interfaces to the trader (§3.3) into a group. Provided that nothing else has been done, this will be the two counters just created.

At this stage, the servers (or more precisely, their stubs) know that they are a group, but the client stubs have not been updated. If the ++ operation is invoked, the client stub that was used will be updated, but only one of the counters will be invoked because the quorum mechanism has not yet been implemented. A futher invocation through that client stub invokes both members and the inconsistency is revealed both on the display and by a collation failure. As shown in figure 4.2, the counters have been updated by the operation that filaed to collate.

```
% $c1 ++
2
% $c1 ++
Collation failed
%
```

Figure 4.2: Consistency problem

This simple system happens to have the property that invoking the ++ operation through the other client stub will make the states of the servers consistent and also bring the stub up to date.

```
% $c2 ++
3
% $c1 ++
4
% $c2 ++
5
%
```

As can be seen from the example, there are now two distinct client stubs for the group service, both of which invoke both members.

4.2.7 Issues to be explored

Information about the group and its members is scattered throughout the system. The prototyping technology can be used to explore many issues.

Out of date client stubs may have propagated arbitrarily far, and new, already obsolete, stubs can be generated by the parameter passing mechanism. The issues of obtaining sufficiently fresh information without excessive overhead need to be explored.

Offers in traders may become obsolete when group membership changes. This may be an aspect of the client stub issue, but traders do not ususlly invoke the services that they describe so may be implemented so as to avoid creating unnecessary stubs.

In the example above, two distinct services were merged into one after both had been used. What does this mean for the clients of those services? Should this be discouraged, or even prohibited? If so, how is the prohibition to be enforced?

Both services in the example had been exported with different properties. After the merger, both offers would lead to the same service despite the different properties.

If services can merge, can a group split into a number of distinct services with part of their history in common? It is easy enough to implement this by telling the server stubs the members of their partition. This would trigger the client stub update, but what ought it to do in this case? The implementation in the prototype has a race condition where the partition that gets to the client *last* 'wins' and becomes the server; the clients would probably end up using the most remote or slowest service.

These issues have been exposed by the very simple experiments conducted already. Further issues will be raised when the use of dependency information to give more efficient management of redundancy is explored.

5 Interpreter

5.1 Base technology

The interpreter consists of the following parts:

- Tcl base language
- Tk graphics extensions for Tcl
- [incr Tcl] object-oriented programming extension for Tcl
- Tcl-DP distributed processing extension for Tcl

Some combinations of these parts are publicly available by anonymous FTP; they are supplied with automatic configuration scripts which makes them easy to install on many platforms.

An interpreter including all of these parts has been constructed for the dependability prototyping, and this too has an automatic configuration script. This interpreter is described in §5.2.

Another extension to Tcl that is in use at APM, and which may be added to the combined interpreter in the future is:

- expect interface to interactive programs

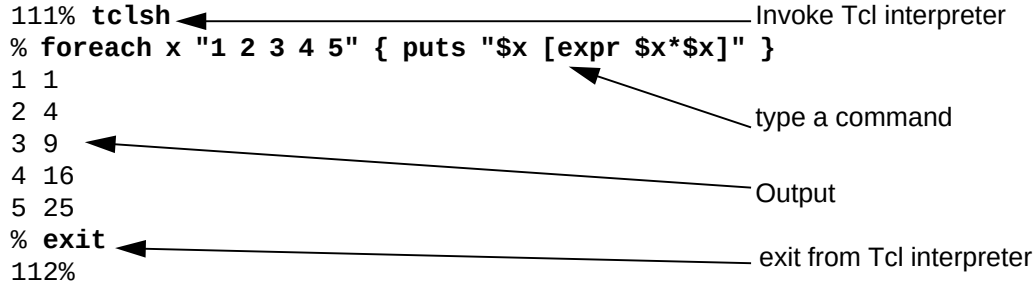
The various parts, and the combinations which are available are described below.

5.1.1 Tcl

Tcl - Tool Command Language - is a script language, originally designed to be embedded in applications implemented in C.

The current version includes a 'shell' application that allows the language to be used interactively at a terminal, or for script files. Figure 5.1 shows a very simple example of interactive use of Tcl.

Figure 5.1: Tcl example

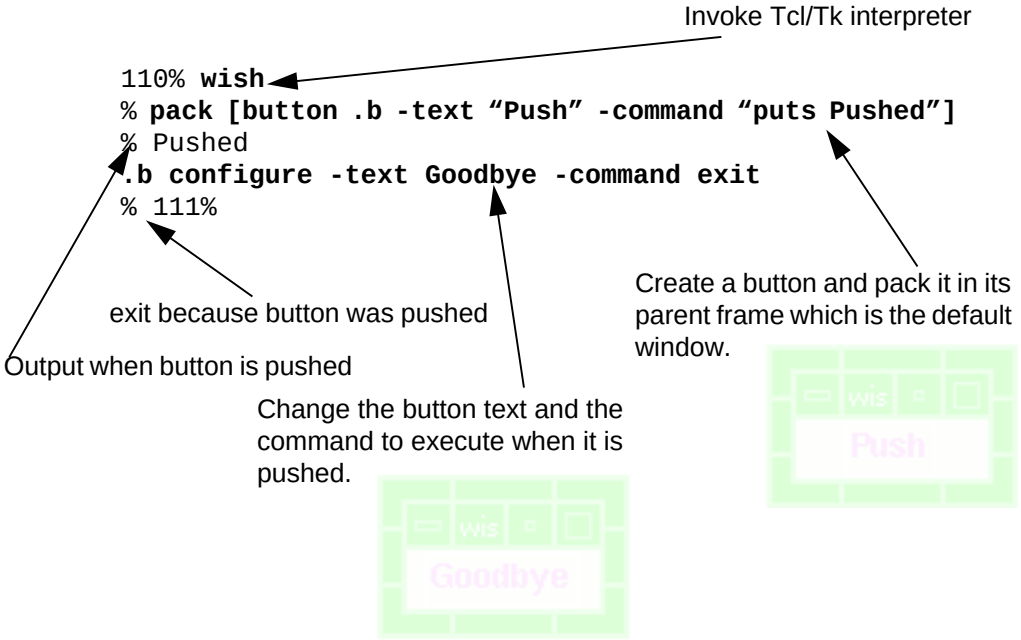


5.1.2 Tk – Graphics extension to Tcl

Tk is an extension to Tcl that provides high-level support for graphical user interfaces.

Tk includes a 'window shell' that allows a variety of graphical objects to be created, configured, positioned in frames in windows, and destroyed. Figure 5.2 shows interactive creation and reconfiguring of a button.

Figure 5.2: Tk example



5.1.3 [incr Tcl] – Object oriented programming extension to Tcl

[incr Tcl] [McLennan 93] is to Tcl what C++ is to C, both in name and purpose. It adds object oriented programming features to Tcl, in particular, methods, classes and inheritance. It turns object instances into Tcl commands that take a method name as their first argument. Figure 5.3 shows a very simple example.

Figure 5.3: [incr Tcl] example

```
112% itcl_sh
% itcl_class counter {
 constructor {config} {}
 method config {config} {}
 method value { } { return $count }
 method ++ { } { return [incr count]}
 protected count 0
}
% counter c1
c1
% counter c2
c2
% c1 ++
1
% c1 ++
2
% c2 ++
1
% exit
113%
```

Invoke [incr Tcl] interpreter

create a class

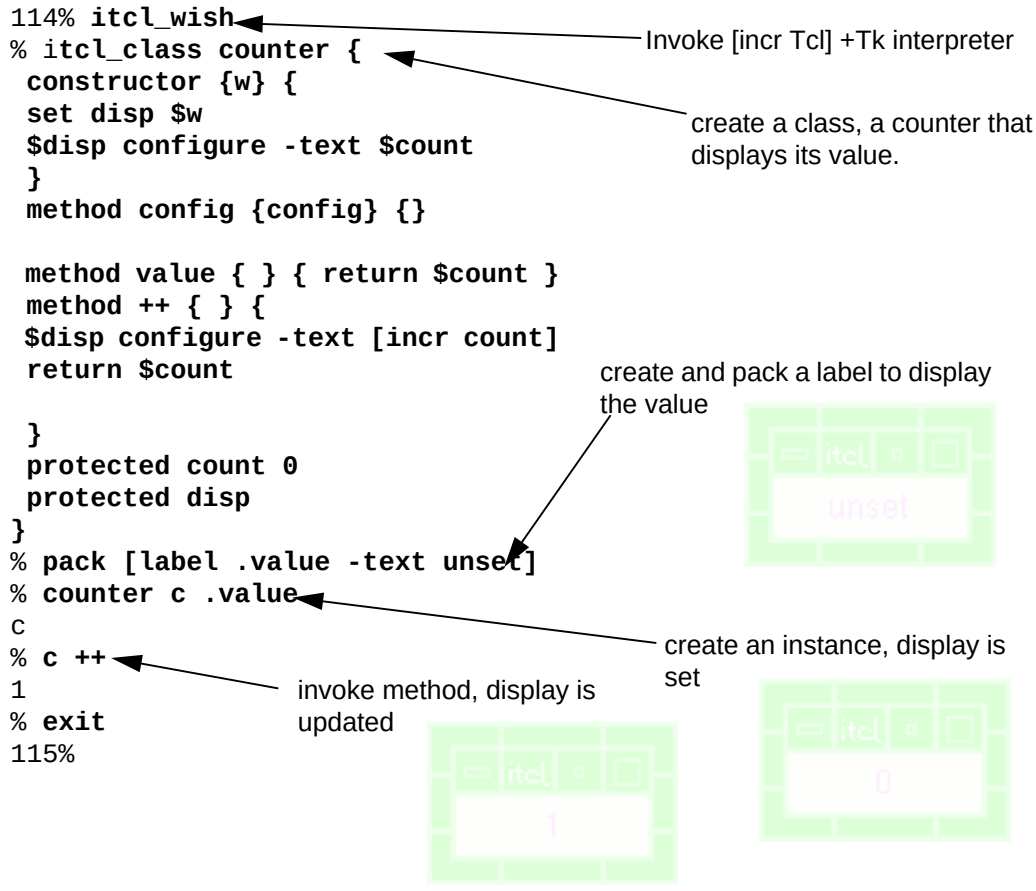
create an instance

create another instance

invoke methods on the instances

The [incr Tcl] package comes with an interpreter that includes Tk as well as [incr Tcl]. Figure 5.4 shows an extended version of the previous example that uses both [incr Tcl] and Tk.

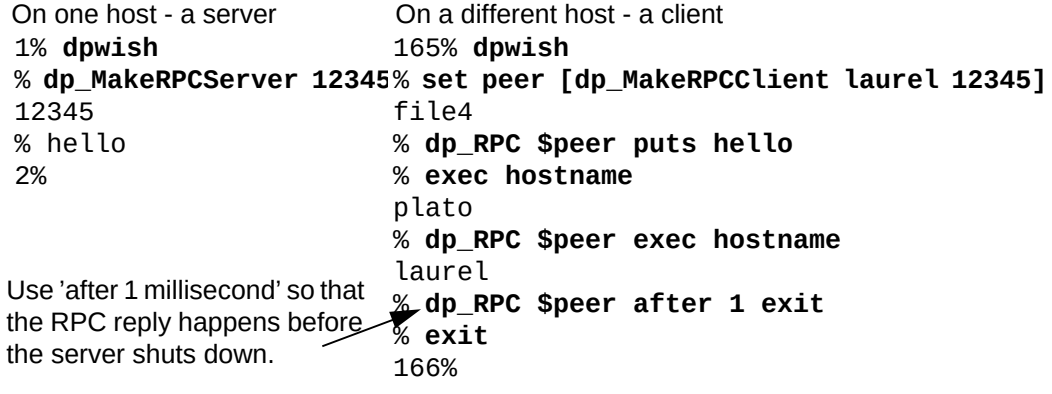
Figure 5.4: [incr Tcl] and Tk together



5.1.4 Tcl-DP – Distributed Programming extension to Tcl

Tcl-DP provides TCP and IP connection management, a basic RPC facility, and something the authors describe as "distributed object support" which is best avoided. The RPC system is of the "open a connection; make RPCs over the connection" variety, and is best considered a low-level building block.

Figure 5.5: This is a figure



5.2 [incr Tcl-DP] – Distributed Object Oriented Tcl

The interpreters described in §5.1 are freely available packages which can be retrieved over the Internet.

[incr Tcl-DP] was put together from the parts of those freely available packages as a prototyping environment for exploring programming issues for dependability. The intention is to implement an example in order to explore dependability issues.

The interpreter itself supports the commands of Tcl, Tk, [incr Tcl] and Tcl-DP. Chapter 2 describes the library that takes these basic elements and turns them into a distributed object-oriented programming environment.

5.3 Summary of interpreters

5.3.1 Publicly available interpreters

5.3.1.1 Interpreters described above

```
tclsh     : Tcl
wish      : Tcl + Tk
itcl_sh   : Tcl      + [incr Tcl]
itcl_wish : Tcl + Tk + [incr Tcl]
dpwish    : Tcl + Tk              + Tcl-DP
```

5.3.1.2 Other interpreters in use at APM

```
expect    : Tcl + expect
expectk   : Tcl + Tk + expect
wwwish    : Tcl + Tk + commands to drive the http library!
```

5.3.2 Interpreters for dependability prototyping

5.3.2.1 Implemented interpretrers

```
idpwish   : Tcl + Tk + [incr Tcl] + Tcl-DP
```

5.3.2.2 Interpreters which may be implemented if needed

```
idpexpectk: Tcl + Tk + [incr Tcl] + Tcl-DP + expect
idpexpect : Tcl      + [incr Tcl] + Tcl-DP + expect
idptclsh  : Tcl      + [incr Tcl] + Tcl-DP
```

References

[McLennan 93]

McLennan, M. J., “[incr Tcl] – Object Oriented Programming in Tcl”, included as Intro.ps in file://harbor.ecn.purdue.edu/tcl/extensions/itcl-1.3.tar.Z

[OUSTERHOUT 94]

Ousterhout, J. K., “Tcl and the Tk Toolkit”, Addison-Wesley 1994.