Iclient/Iserver: Distributed Objects using [incr Tcl]

Lee F. Bernhard
Bell Labs Innovations for Lucent Technologies

Iclient/Iserver: Distributed Objects using [incr Tcl]

Lee F. Bernhard
Bell Labs Innovations for Lucent Technologies
Now at Scriptics Corporation
lfb@scriptics.com

Iclient/Iserver is a simple distributed object framework for [incr Tcl] applications that enables its clients to synchronize activities and share information. Using Iclient/Iserver, clients can access objects living on a remote server tranparently, making building client/server applications both easy and intuitive. Iclient/Iserver is conceptually similar to the widely used CORBA standard, but is much simpler, intended for building smaller, client/server applications where the cost and complexity of a CORBA implementation cannot be justified.

In this paper I describe the use of Iclient/Iserver for sharing server objects among clients. I explain the underlying architecture and implementation of the distributed object system. I conclude by illustrating how to use Iclient/Iserver to build a simple networked version of the card game Hearts.

Introduction

Since the Tcl/Tk core added sockets and safe interpreters, using Tcl/Tk to create networked applications is very simple. This simplicity opens up a realm of applications in which users can communicate and share information using Tcl/Tk over a network. With the Tcl/Tk Plugin[1], programmers can bring a networked application right to the user's desktop via a web browser.

Networked applications are usually developed in the client/server paradigm, in which a server provides central services, and client programs access these services over a network. A server might control a database, and clients might communicate with the server to query and manipulate the data. Or a server might host a multiplayer card game, and clients might connect in to play the game.

In all of these applications, the server centralizes shared data. It synchronizes the actions of the clients, so that, for example, one client can't access a data record while another is updating it. The server also broadcasts significant changes, so that when one client modifies a data record, other interested clients automatically know about it.

A significant problem in writing client/server applications lies in defining the application programming interface (API) between the server and the client. The server has some subset of procedures that it wants to publish to the client to allow the client access to its services; the difficulty lies in providing an API that publishes these procedures in a way that is both simple and manageable in terms of size and complexity. In a networked card game, there might be a procedure to query the list of games, a procedure to join a game, a procedure to query a hand, a procedure to play a card, a procedure to check the score, and so on. Even for a simple application like Hearts, the number of access procedures can quickly get out of hand.

Iclient/Iserver greatly simplifies the development of client/server applications by allowing the programmer to implement clients and servers with Tcl/Tk and the [incr Tcl] object system[2][3]. With just a few lines of code, programmers can set up an object server that publishes its objects to any connected client(s). In a stand-alone [incr Tcl] script, class methods provide the major interface for performing operations and manipulating data. By publishing its objects, the server allows clients to use the same interface that the server script uses to access its objects. The server's existing object system becomes the API for clients to access server services, eliminating the bother of writing procedural wrappers around each server object.

Iclient/Iserver is not the only solution to provide distributed objects; perhaps the most widely recognized solution to providing distributed objects is the Common Object Request Broker Architecture (CORBA)[4][5], a distributed object standard designed by the Object Management Group (OMG) after eight years of discussion and collaboration. It uses a model of an "object bus" that clients use to access remote objects. CORBA builds upon the successes and services of previous technologies like RPC and TP monitors. It is an ambitious framework, allowing clients to access objects implemented in a different language than the client program. But its ambition also makes it a complicated framework to learn and work with; the complexity is hard to justify when creating smaller client/server applications. Additionally, CORBA's goal of allowing multiple languages to share components requires a translation layer (IDL) that isn't necessary, for example, in building small Tcl/Tk applications.

There is already support for procedural client/server development in the Tcl/Tk community. The Tcl-DP extension [6][7] makes it easy to set up RPC-style client/server applications with a procedural API. It includes some support for simple, slot-based structures. The GroupKit extension [8][9] is another RPC-based solution with slot-based structures, designed with collaborative applications in mind.

There are a number of extensions that allow Tcl scripts to access CORBA objects and services. These extensions make it easy to write Tcl scripts that control CORBA distributed components, and several have facilities for creating Tcl structures that other CORBA clients can access. They don't work seamlessly with [incr Tcl]; they require class descriptions written in a neutral language format, IDL.

Iclient/Iserver is similar to these packages in some respects, occupying a space somewhere between the convenience and simplicity of Tcl-DP and Groupkit and the robust object support of CORBA. It lets both clients and servers take full advantage of object-oriented technology. Instead of using a simple, slot-based object model, Iclient/Iserver uses [incr Tcl] to provide a full-featured, class-based model. Classes can encapsulate data and related operation, with support for public, protected, and private members. [incr Tcl] also supports single and multiple inheritance, so that classes can share functionality.

Iclient/Iserver harnesses [incr Tcl]'s existing code base and adds a quick, easy way to distribute objects. With Iclient/Iserver, programmers can concentrate on designing classes that provide core functionality; they can add the client/server capability almost as an afterthought.

Counter: A Simple Example

Suppose you have a `Counter` class with an object named `foo`. This object contains a number, which programmers can increment and query using the methods `bump` and `value`. An object like this could help clients to generate serial numbers for processes, orders, transactions, etc. Programmers can create the object on the server and make it available to clients with the following code:

```
package require Iserver
class Counter {
    inherit ::iserver::Distributed
    method bump {} {return [incr count]}
    method value {} {return $count}
    private variable count 0
}
Counter foo
iserver::listen 8066
```

The server begins by initializing the `Iserver package`, thereby creating commands and base classes that the server needs. It then defines an [incr Tcl] class `Counter` with two methods `bump` and `value`. By inheriting from the `Distributed` class, the `Counter` class is able to share its objects with clients. The server creates a `Counter` object called `foo`, and then opens a socket on port `8066` and waits for clients to connect and start using `foo`.

Now suppose there are two clients that would like to use the server's `foo` object. Each client needs to load the `Iclient` package, and then connect to the server. It does so by creating a `Server` object, and telling it to connect to the server listening at port `8066` on the machine `sercial.micro.lucent.com`.

```
package require Iclient
set serv [::iclient::Server #auto \
  sercial.micro.lucent.com 8066]
$serv resolve class Counter
$serv resolve object foo
foo bump
```

`Now that the client is connected to the object server`, it can attach itself to the `Counter foo`. To do this, the client resolves the names of both the class, `Counter`, and the object, `foo`. In other words, the client has gone to the server and created its own, local copy of `Counter` and `foo`, which it can use to access the actual, server entities of the same names. All the client needs to do is call the `bump` method on its object, and Iclient will take care of running `bump` on the server object `foo`.

The local copy of `foo` is merely a client-side stub. Methods are not truly implemented on the client side; instead, the client methods serve as an interface for calling corresponding methods on the server object `foo`. A client stub can also be considered conceptually as a local, client-based, reference to a server object.

Now imagine that the clients want to use the Counter foo in a simple GUI. A label will store the current value of foo, and a button will allow the client to bump the value of foo. If one client changes the value of foo, the GUI of the other client should update to show the new value as well.

```
...
button .bump -text "Bump" -command {
  foo lock {
    .count configure -text [foo bump]
  }
}
pack .bump -side left -padx 6 -pady 10
label .count -text [foo value]
pack .count -side left -padx 6 -pady 10

foo watch lock {
  .count configure -text [foo value]
}
```

This example uses two object services of Iclient/Iserver: lock and watch.

When pressed, the .bump requests a lock on foo. It then increments its value by calling the bump method.

Interprocess communication presents a host of concurrency problems that occur when two clients try to manipulate the same resource at the same time. In Iclient/Iserver, most method calls will be atomic, unless they access the event loop. To be certain of exclusive access to a server object, a client should request a lock whenever modifying a server object.

Locks are especially helpful when a client needs to manipulate a server object with multiple method calls, where the entire series of operations needs to run atomically. By acquiring a lock on a server object, a client gains exclusive control of that object.

In a slightly more complicated scenario, clients may be using the Counter object to keep track of the current bid price in an auction. Clients will check the current price, and if it falls below maxBid, they will bid by incrementing the Counter foo.

```
if {[foo value] < $maxBid} {
  foo bump
}
```

What makes this example different is that the clients want to check the state of an object, make a decision based on that state, and then call another method on the object to change it. Without locks, there is a classic race condition where it is possible for a client to make the wrong decision about whether to bid.

Imagine that the bid price is one less than maxBid for each client. The object server processes client requests in the order they are received. If both clients check the value of foo, they will both receive the same value and decide to bump the Counter. When this occurs, one of the clients will exceed their maximum bid, because the state of foo will have changed between the time the client checked its value, and the time it tries to bump the foo.

Locks solve this problem nicely by allowing each client to check foo and bump its value without interruption from the other client. immediately:

```
foo lock {
  if {[foo value] < $maxBid} {
    foo bump
  }
}
```

This is guaranteed because once the first client acquires the lock on foo, it is the only client allowed to access foo until the lock is removed. If another client tries to access foo while the first holds the lock, it will have to wait until the lock is released.

By default, clients will ten seconds for the lock; if that time expires, the lock method raises a Tcl error. Clients can adjust the timeout factor by providing a value, in milliseconds, for the -timeout option of the lock method. A timeout of zero indicates that the client should give up waiting if it cannot obtain the lock. If the client may need to wait for a noticeable period to obtain a lock, it will likely wish to display a watch cursor, or some other GUI indication that the client is busy. The lock method provides two more options, -suspend and -continue to allow the client to run a script immediately before waiting for the lock, and immediately after the lock is obtained. The Counter clients may use this to display a message in the label .count as they wait for the lock:

```
foo lock {
  .count configure -text [foo bump]
} -timeout 20000 -suspend {
  .count configure -text “Acquiring lock”
} -continue {
  .count configure -text “Lock acquired”
}
```

One more detail remains to be explained: when one client bumps the value of foo, the other clients need to know that foo has changed in order to update their displays. Clients can watch objects, and register a callback that will run when a particular event occurs to the object. This is similar to a variable trace or to a binding on a Tk widget. Keeping track of an object being locked is rather straightforward. The clients can `watch` for `foo` to be locked; when that occurs, they know that `foo` has changed, and they can call the `value` method to learn the new value of the `Counter`. Clients can `watch` any object, looking for well-defined or custom-defined events to occur. Well-known events include `lock`, `destroy`, and `resolve`. `Distributed` objects can report custom events by using the `report` method, as in the following:

```
class Counter {
  inherit ::iserver::Distributed
  method bump {} {
    report "bump"
    return [incr count]
  }
  ...
}
```

The last services that Iclient/Iserver provide are the ability to restrict use of a server object to a subset of connected clients. An object server gives each of its clients an unique id, and can use this id with the `Distributed` class's `restrict` method. Here a server restricts use of the object foo to a single client with client id `client1`:

```
foo restrict {client1}
```

Architecture

Applications written using Iclient/Iserver are constructed with a client/server architecture where the server program provides central services for the client programs connected by a network. The server helps the clients to interact by holding shared information and synchronizing the clients' access to this information.

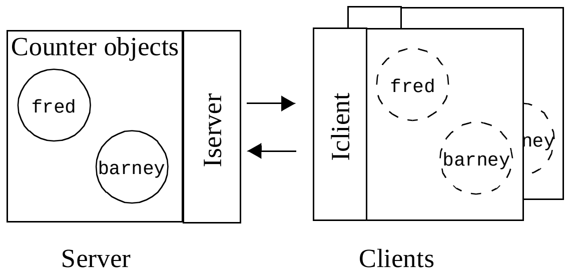
In any client/server architecture, an API mediates between clients and server services. In the remote procedure call (RPC) approach, the server program publishes a set of commands that clients use to communicate with the server. An RPC approach works well for procedural applications, such as vanilla Tcl scripts, where command provide the only interface for performing operations.

An [incr Tcl] script uses classes to encapsulate data and related commands into a single entity. Each class has a well-defined set of methods that act as the interface for working with objects of that class. By invoking these methods, it is possible to access and manipulate the data contained by the object.

Turning an [incr Tcl] script into a client/server application should be a process that keeps the interfaces defined by the classes intact. A RPC approach does not work well with [incr Tcl] scripts because it requires wrapping each method with a procedure. there to be a procedural wrapper for accessing the methods of an object. Adding a procedural communication mechanism to an object-oriented design breaks up the design of classes. At best, it is inelegant. At worst, it is tedious to set up and confuses the interface created by the class methods. It would be much better to allow clients to access server objects in the manner that the server manipulates its object: by calling the methods defined for the object.

That is the goal of Iclient/Iserver. The server uses [incr Tcl] to create classes and objects with well-defined interfaces. Iserver provides a base class, `Distributed`, that publishes derived classes and their objects, so that clients connecting to the server can use those objects across a network socket.

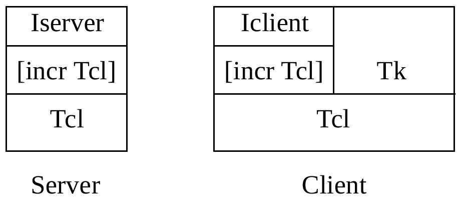
Iclient provides the client access to server objects by creating local copies, or stubs, which act as references to the server objects. When a client access a client stub, the stub encapsulates the server communication required to access the corresponding server object.



Server objects are visible on Clients

It is possible for a client to be connected simultaneously with multiple object servers. Iclient provides a class `Server` to help client scripts to keep track of a given object server connection. When a client wants to connect to a new object server, it creates a new instance of this `Server` class and feeds it information about the hostname and port number of the object server. The methods of the `Server` class provide a secondary API that clients use to access an object server, to create client stubs, create new server objects, destroy server objects, and inquire about objects available on the server.

Tcl Package Architecture of Client and Server



Iclient and Iserver are currently implemented as Tcl scripts that rely upon [incr Tcl] to provide classes and objects. The first alpha of Iclient/Iserver used [incr Tcl]'s namespace facility, but the revised version uses [incr Tcl] 3.0 and Tcl 8.0 namespaces. Iclient and Iserver use namespaces to encapsulate the classes and commands that they introduce.

Distributed Object Model

This section describes in detail how the Iclient/Iserver distributed object model works. It covers the two ways clients can create client stubs for server objects. It discusses the underlying commands and classes that the stubs use to communicate with the server. It also describes how sockets, fileevents, and safe interpreters form the primitive communication framework.

The central task for Iclient/Iserver is to allow clients to access server objects in a manner consistent with accessing ordinary [incr Tcl] objects in a stand-alone application. To do this, a server object must appear as if it lived locally, on the client. It must also have the same methods as the server object, and the client must be able to call those methods as with ordinary [incr Tcl] objects.

To accomplish this, Iclient/Iserver literally creates a client-side object to act as a stub for the server object the client would like to use. These stubs are [incr Tcl] objects; therefore, the client requires a class declaration to produce the stub. The `Server` class that servers as an API to access an object server provides a method `resolve` that resolves the binding between the client stub and the server object.

Iclient/Iserver provides both implicit and explicit mechanisms for resolving references to server objects. In the implicit model, the client code does nothing to resolve references to server objects. The client merely connects to an object server, and starts using well-known objects. To return to the `Counter` example, suppose a client runs the following code:

```
package require Iclient
set serv [iclient::Server #auto \
  sercial.micro.lucent.com 8066]
foo bump
```

The client has connected to an object server but has done nothing to link the server object `foo` to the client. The client proceeds anyway, calling the `bump` method of `foo`. The Tcl parser looks for a command named `foo` but cannot find one defined. Ordinarily, the parser would raise an error saying that the command `foo` does not exist. But Iclient has changed the way Tcl handles unknown commands by introducing its own handler into the built-in `unknown` command.

This handler knows about the object servers that the client is attached to, and it asks each object server if it can `identify` the name `foo`.

```
$server ask identify foo
```

The `ask` command is a primitive to Iclient and Iserver that allows a process to send a Tcl command to another process and wait for the result. In this case, the client sends the script "`identify foo`" to the server, who will evaluate the `identify` command. This command looks to see if `foo` is either the name of a distributed class or a distributed object on the server. The server determines that `foo` is an object, and sends a reply to be evaluated in the client:

```
respond 1 0 "object Counter"
```

The respond command is used to tell the client that the server has a formulated a response to one of its `ask` requests. It reads the return code and the return value from the server, and learns that `foo` is an `object` of class `Counter`. The client knows that `foo` is a `Counter` object, and tries to create it on the client side:

```
Counter foo
```

Unfortunately, the client has also never seen the command `Counter` before. The `unknown` command runs again, this time looking for `Counter`, and asks the server if it knows of a name `Counter:`

```
$serv ask identify Counter
```

This time the server replies that `Counter` is a class. The client asks the server to supply a class stub for the `Counter` class:

```
$serv resolve class Counter
```

```
Counter foo
```

The server must now generate a stub for the class `Counter`, and pass the stub for the client to use. This involves creating a new class declaration that has only the public methods of `Counter`. These stub methods are not implemented as they are in the client class declaration; instead, each is a wrapper around an interface that runs the same method on the server object. The stub class declaration created for the `Counter` class might look like this:

```
class Counter {
  method bump {args} {
    return [eval $server ask object \
      invoke [namespace tail $this] bump \
      $args]
  }
  method value {args} {
    return [eval $server ask object \
    invoke [namespace tail $this] value \
    $args
  }
  ...
}
```

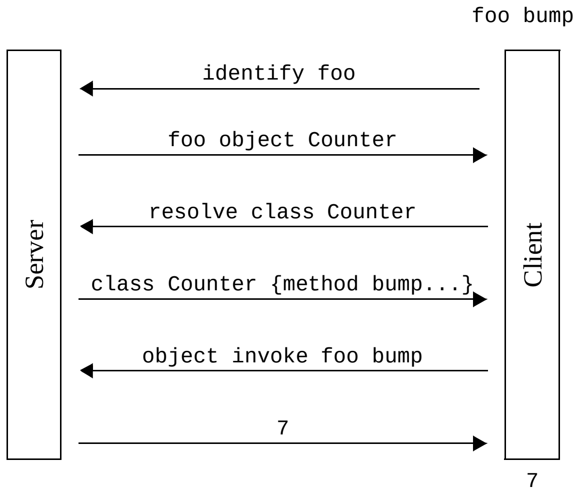
Now the client has a stub for the `Counter` class. It can now try to resolve a reference to `foo`.

```
$serv resolve object foo
```

The client asks the server to help it resolve a reference to foo. This causes the client to create a client stub called `foo` using the class declaration previously generated by the server. With the stub in place, the client can now invoke methods on the stub; when it does so, the stub will ask the server to invoke its own object named `foo` and will use the response from the server as its return value.

This rather detailed chain of events is illustrated in the figure below. What is important to realize is that all this detail was handled by the `Iclient` package; the client script set this into motion by the innocent looking call of `foo bump`. The entire process of resolving is transparent to the programmer, who needed only to create an object on the server, and use that object on the client.

The advantage to this system is that the client code does not have to declare server classes or objects explicitly before using them. Iclient performs late binding to the server objects, doing it only when necessary, and handles all the details of class and object stub generation. But implicit resolving requires Iclient to search through all the object servers connected to a client--usually this



Highlights of Implicit object binding;
Client accesses Server object `foo` for the first time.

represents only a single server, but that does not have to be the case.

To solve these problems, Iclient also offers an explicit interface for generating client stubs. A client can request a stub from a `Server` object, by calling that object's `resolve` method.

```
set serv [Server #auto sercial 8066]
$serv resolve class Counter
$serv resolve object count mycount
```

Above, a client has created a `Server` object to connect to an object server. It then resolves two references, one to the server class `Counter`, and another to the server object `count`. The process of resolving references is the same from this point on; the server helps the client to generate a class declaration and a client stub. The difference is that the client script programmer has explicitly resolved the references before trying to access server objects.

Iclient/Iserver primitives

Iclient/Iserver relies upon a handful of primitives to organize message passing between clients and servers. These primitives include `ask`, `tell`, and `respond`. Whenever a client communicates with a server, or a server communicates with a client, one or several of these primitives are used.

For example, when a client asks a server to create a stub, the client sends a command to the server including the request. The `ask` primitive allows the client to send a Tcl command to the server, evaluate that command on the server side, and learn the result. It works by sending a Tcl command through the client's socket to the server, and supplying the server with a callback. The server uses a fileevent to notice that there is traffic from the client, and reads the command off the socket. It evaluates the command within the limited context of a safe interpreter, and generates a return value. To report the result of the operation, the client sends the client's callback over the socket, along with the server's result. The client reads the callback, evaluates it in its own safe interpreter, and returns the server's result.

```
respond 1 0 “object Counter”
```

The respond command is used to tell the client that the server has a formulated a response to one of its `ask` requests. It reads the return code and the return value from the server, and learns that `foo` is an `object` of class `Counter`.

A third primitive, `tell`, allows a command to be run asynchronously on the other side of the socket. For example, a client can `tell` a server to invoke the `bump` method on object `foo`. The client won't wait for a response; the command will run on the server side whenever the server has a chance to process it. Meanwhile, the client has moved on to other things.

Iclient/Iserver makes heavy use of core Tcl sockets, fileevents, safe interpreters, and the unknown handler to implement distributed objects. Other sources [10] discuss how to connect two processes with Tcl sockets so that each can send and evaluate Tcl commands on the other. Iclient/Iserver follows this approach, and extends it for use with objects.

The server evaluates commands sent from its clients in a safe interpreter, to prevent malicious clients from evaluating destructive commands in the main interpreter. Imagine if a client ran a troublesome command:

```
$server ask exec rm -rf .
```

The server evaluates the command `exec` in its safe interpreter, but `exec` is not defined there. Instead of happily eating up the filesystem, the command raises an error, and the server is safe from harm.

All of the commands that clients call to identify, resolve, create, manipulate, and destroy server objects are defined in the main interpreter, to allow these commands to access the objects there. To allow the clients to call them, the server creates aliases in the safe interpreter, so that when a client asks to run the command `identify`, it does so in the main interpreter of the server.

Application: Hearts

In this section, we will build a game of Hearts that allows four players across a network to meet and play the card game Hearts.

Hearts is a card game usually played by four or more players around a table. Our game will allow players to meet up at a hearts server running in some well-known location, choose a game to join, and then play a game of Hearts. Each player will need to have his own interface with which to play the game.

A client/server approach is natural for this application. There are multiple players, each of whom have their own cards that they want to hide from the others. Each player will be running their own client to play the game, but will need to coordinate their play with the other clients. A Hearts server will manage the deck of cards, synchronize the actions of the clients, and control the flow of the game.

One possible approach to implementing this application is to build a stand-alone version first, that runs on a single machine and lets players take turns playing. There are a lot of details to organize for each game of Hearts--players playing cards, checking their score, organizing their hands--and the engine needs to keep track of what cards have been played, decide whether a player has selected a valid card, etc. On top of this, there are many concurrent games of Hearts being managed by the same server. This problem is neatly solved using an object approach, to help encapsulate each of the Hearts games being played, and to establish a convenient interface between the client and server parts.

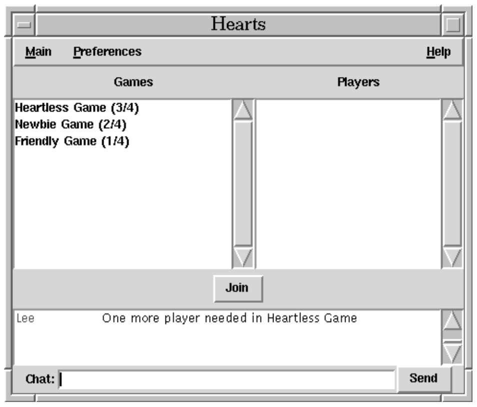
The `Gamemanager` class manages `Game` objects, helping clients to create, join, and quit new games. It also allows clients to register as players, with names and email addresses.

The server initializes itself by defining all the classes it will use during its lifetime, and by creating a well-known `Gamemanager` object, `gm`.

```
package require Iserver
class Gamemanager {
  method games {}
  method players {}
  method newplayer {option args}
  method newgame {option args}
}
class Game {
  method players {}
  method join {}
  method quit {}
  method cards {}
}
...
Gamemanager gm
iserver::listen 8181
```

This is the main script for my Hearts server application; the bulk of the work comes in creating the classes that the server will use to manage games of Hearts.

When a client connects to the Hearts server, it displays a GUI showing descriptions of the various games in session, and the players taking part in the selected game. To learn this information, the client uses the well-known `Gamemanager` object `gm`.



Heartless Game (3/4)
Newbie Game (2/4)
Friendly Game (1/4)

Lee   One more player needed in Heartless Game

Hearts client game selection screen

The following block shows how a client might fill a listbox `.games` with description of the `Game` objects that the `Gamemanager gm` knows about. The binding places the selected `Game`'s players in the listbox `.players`. The button `.join` asks to `join` a `Game`.

```
proc show_games {} {
  global games
  .games delete 0 end
  set games [gm games]
  foreach game $games {
    set desc [$game description]
    .games insert end $desc
  }
}
proc show_players {game} {
  .players delete 0 end
  set players [$game players]
  eval .players insert 0 $players
}
bind .games <ButtonPress-1> {
  set idx [.games curselection]
  set game [lindex $games $idx]
  show_players $game
}
button .join -text “Join” -command {
  $game lock {$game join}
}
```

This block of client code begins by asking the `Gamemanager` object `gm` for a list of `Game` objects. The names of the game objects will be rather dull strings like `game1` or `game7`. The client calls the `description` method of each `Game` object to get a short description of the `Game`, placing this in the listbox `.games`. When the user selects a description in the listbox `.games`, the binding calls the `players` method of the corresponding `Game` object, and uses those names to fill the `.players` listbox.

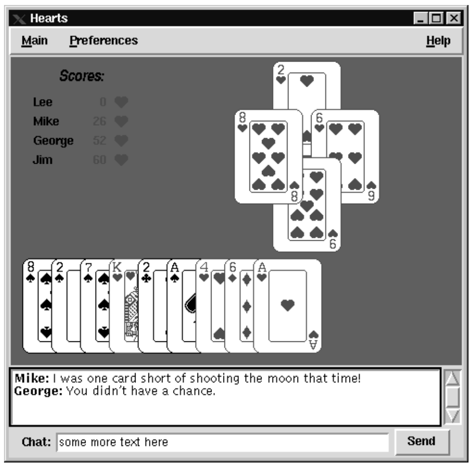
The `.join` button allows a player to join a `Game` that is forming. Since this alters the state of that `Game`, the button command requests a lock before joining. All the players in that game will want to know that a new player has joined. They automatically learn this by `watch`ing the `Game` object; when the lock is requested, the clients ask the `Game` object for the new list of players, and update their listboxes.

```
$game watch lock {
  show_players $game
  show_games
}
```

Now the GUI changes as players enter and leave games. The client also monitors the Gamemanager to `watch` for `Game` objects being added:

```
gm watch lock {
  .games delete 0 end
  set games [gm games]
  foreach game $games {
    .games insert end [$game description]
  }
}
```

Suppose our player has selected a `Game`, and it has filled with players. It is now time to begin a game of Hearts. The view of the available games is replaced with a canvas showing the game table, with the player's cards at the bottom, and a view showing the cards that others have played in the center. The player can see the current score of the game at the upper-left corner. A chat box at the bottom allows the player to send messages to the other players in the game.



Mike: I was one card short of shooting the moon that time!
George: You didn't have a chance.

some more text here

To generate the hand, the client calls the `cards` method of a `Game` object. Assume for our purposes that the name of the `Game` object for this game is stored in the variable `game`. The following client code requests a list of cards in the player's hand, and draws them on the canvas by calling a command on the client.

```
card_draw [$game cards]
```

When the player is ready to play a card, they drag the card to the center of the table. The game of Hearts is played by following an order of play. One player leads by playing a card on the table, then play moves around the table in a circle. If our client tries to play a card, the server must first decide if it is that player's turn.

The `Turn` object keeps track of the order of play for a given trick, and prevents players from playing out of turn. Players invoke the `play` method of the `Turn` object in order to play a card.

To do this, the client requests a lock on the `Turn` object. The `Turn` class exists primarily to regulate whose turn it is, and to allow only the appropriate player to play a card. To accomplish this, the Turn object establishes a set of locks on itself for each of the players playing the game. These lock requests enter a queue, and are entered in turn order. If a client tries to access the turn object prematurely, then that client must wait for the players ahead in the play order to play their cards; as soon as they are finished, then our client acquires the lock and plays a card.

Client Code

```
$turn lock {
  $turn play DK
}
```

Server Code

```
class Turn {
  inherit ::iserver::Distributed
  private variable order;
  method newturn {}
  method discard {}
  private method accept {card}
  method play {card} {
    set player [::iserver::current]
    set turn [lindex $order]
    if {$player == $turn} {
      accept $card
      report $discard
      return
    } else {
      error "Not your turn"
    }
  }
}
```

The `Turn` object controls the order of play. With every trick, the `Turn` object calculates the order of play based on the rules of Hearts. When a player attempts to `play` a `card`, the `Turn` object asks Iserver who this client is, and checks to see if it is the client's turn. If it is, the `Turn` object `accept`s the `card`, and reports a `discard` event to let the other clients know to redraw their canvases. If the player should not be able to play yet, the `Turn` object does not let the player `play.`

All of the clients are going to be interested in the cards that the other players have played, in order to draw them in the center of the canvas. To watch for players playing cards, each client watches the `Turn` object. When another client accesses the `Turn` object to play a card, the server notifies each of the clients, who then learn which cards were played and draw them:

```
turn watch discard {
  set cardsPlayed [$turn discard]
  discard_draw $cardsPlayed
}
```

Notice that the script a client runs in response to watch event can do anything; it does not need to limit itself to accessing just the object being monitored. The client is watching the turn object to see when a card is played; once it is notified, it calls the `discard` method of the `Turn` object to learn which cards were played, and then draws them using the `discard_draw` command.

There are situations in which it is more convenient for a server to broadcast commands to its clients than for the clients to register interest in an event using `watch`. Players communicate with one another by using an object of class `Chat` that lives on the server. When players wish to talk to the others in the game, they call the `mesg` method of the `Chat` object, and pass their message along. The easiest way for the `Chat` object to communicate the message along to the clients is to keep a list of listening clients, and then remotely invoke a procedure on the client side to

Server Code

```
class Chat {
  method listen {} {
    set client [::iserver::client]
    ::iserver::list_add listeners \
      $client
  }
  method talk {mesg} {
    set speaker [::iserver::client]
    foreach client $listeners {
      ::iserver::tell $client rpc \
        chat_mesg $speaker $mesg
    }
  }
  variable listeners
}
```

Client Code

```
$serv rpc chat_mesg {speaker mesg} {
  .chat.text insert end $speaker $speaker\
    "\t$mesg" $speaker
  .chat.text see end
}
```

With distributed server objects, locks, and watch callbacks, the task of organizing a multi-player card game is made easier.

Conclusions

Using class methods as a natural interface between clients and server, distributed server objects allow efficient encapsulation of data and operations.

It allows encapsulation of data and operations and uses class methods as a natural interface between clients and server.

Iclient/Iserver fills a niche between simple Tcl-based remote procedure call systems like Tcl-DP and comprehensive object brokers that follow the CORBA standard. It provides an easy way to construct client/server applications while eliminating the need to write a separate set of remote procedure calls. Distributed server objects, visible and accessible by all connected clients, provide a natural interface for sharing information. The locking and object watching services provide synchronization and event notification to avoid concurrency headaches, including the need to poll continuously to see if the server has changed.

Using Iclient/Iserver protocols, starting to create client/server applications is easy, requiring trivial changes to the server objects that will be distributed and a few lines of code on the server and client ends to establish a connection. After that, the client resolves the objects it wishes to use, and invokes server object methods through its own client stubs.

While the approach is convenient, it can generate a lot of server traffic, since every remote method call requires a network transaction. Future work will examine how best to allow clients to cache additional information in their client to avoid talking to the server to perform read-only operations. This will obviously involve increased complexity in designing classes, and may raise a new set of concurrency issues, but it stands to improve performance and scalability where many clients access a single server.

Future work will also improve security in the system, to allow servers to authenticate connecting clients. Present security is achieved by evaluating socket traffic through a safe interpreter, which restricts the entry-points to the server to the methods of its distributed objects. If the server could authenticate its clients, then it could provide distributed objects that performed more trusted operations with greater security. Finer control over the methods that a given server object publishes to a specific client could also help to create more flexible server classes.

With version 3.0, [incr Tcl] has become a pure extension to Tcl/Tk, allowing vanilla interpreters to use classes and objects by loading [incr Tcl] as a package. With this support, these developers who enjoy the flexibility of scripting and the structure of an object system can use [incr Tcl] without having to build their own custom interpreters. By loading Iclient/Iserver, these same developers can use their existing classes as interfaces and construct client/server applications faster and more easily than ever before.

Acknowledgments

Thanks to Michael McLennan and George Howlett for many useful discussions about client/server architectures and distributed objects, and for their help with this paper. Thanks also to Michael McLennan for his help in designing the interface for Iclient/Iserver software. Finally, thanks to Robin Valenza for losing sleep to edit this paper.

References

[1] Jacob Levy, “A Tk Netscape Plugin,” *Proceedings of the Fourth Annual Tcl/Tk Workshop ‘96*, Monterey, California, July 10-13, 1996.

[2] Michael J. McLennan, “[incr Tcl]: Object-Oriented Programming in Tcl,” Proceedings of the Tcl/Tk Workshop, University of California at Berkeley, June 10-11, 1993.

[3] Michael J. McLennan, “The New [incr Tcl]: Objects, Mega-Widgets, Namespaces and More,” *Proceedings of the Third Annual Tcl/Tk Workshop ‘95*, Toronto, Ontario, Canada, July 6-8, 1995.

[4] Robert Orfali, Dan Harkey, and Jeri Edwards. The Essential Distributed Objects Survival Guide. John Wiley and Sons, 1986. ISBN 0-471-12993-3.

[5] http://www.omg.org

[6] B. C. Smith, L. A. Rowe, and S. Yen, “Tcl Distributed Programming,” *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, Jule 10-11, 1993.

[7] Peter T. Liu, Brian Smith and Lawrence Rowe, “Tcl-DP Name Server,” *Proceedings of the Third Annual Tcl/Tk Workshop ‘95*, Toronto, Ontario, Canada, July 6-8, 1995.

[8] M. Roseman and S. Greenberg, “Building Real Time Groupware with GroupKit, a Groupware Toolkit,” *ACM TOCHI*, March 1996.

[9] Mark Roseman, “Managing Complexity in TeamRooms, a Tcl-Based Internet Groupware Application,” *Proceedings of the Fourth Annual Tcl/Tk Workshop ‘96*, Monterey, California, July 10-13, 1996.

[10] Mark Harrison, Michael McLennan. *Effective Tcl/Tk Programming*. Addison-Wesley, 1998. ISBN 0-201-63474-0.