```
A% dp_address info $a
128.84.154.8 0 {alvin.cs.cornell.edu fw.cs.cornell.edu alvin.cs alvin
fw.cs fw}
```

The return value shows that the IP address of `fw.cs.cornell.edu` is `128.84.154.8`, and that it is known by a variety of names, including `alvin`, `alvin.cs`, and `fw.cs.cornell.edu`. The `dp_address` command can be used to create a function that returns the IP address given a host's name, or vice versa, as shown in the two functions below.

```
A% proc InetAddress {hostname} {
   set addr [dp_address create $hostname 0]
   set x [dp_address info $addr]
   dp_address delete $addr
   lindex $x 0
}

A% proc Hostname {inetAddr} {
   set addr [dp_address create $inetAddr 0]
   set x [dp_address info $addr]
   dp_address delete $addr
   lindex [lindex $x 2] 0
}
A% InetAddress alvin.cs.cornell.edu
128.84.154.8
A% Hostname 128.84.154.8
alvin.cs.cornell.edu
```

By the way, as shown in the next to last line of each of the procedures above, you should execute the `dp_address delete` command to free up the memory associated with an address when you are finished with it.

Learning More

That concludes our tour of the Tcl-DP extension to Tcl/Tk. There are a few other features in Tcl-DP that we haven't discussed, but we have covered the main ones here. We hope you have gained an appreciation for how simple it is to build distributed programs using Tcl-DP and that you have enough background now to explore on your own. To learn more, study the examples in the `examples` subdirectory, post articles to the `comp.lang.tcl` newsgroup, and read the manual pages in the `doc` subdirectory.

Tcl-DP is the result of the efforts of many, many people. Some of the major contributors are Steve Yen, Pekka Nikander, Tim MacKenzie, Lou Salkind, R. Lindsay Todd, Peter Liu, Ulla Bartsich, Mike Grafton, Jon Knight, Gordon Chaffee. You can help, too. If you add a new feature to Tcl-DP or port it to a new platform, please send mail to `tcl-dp@cs.cornell.edu` so that we can incorporate the changes into the source.

can be set using `dp_socketOption`:

- **Loop Back**: When a message is sent to a group address, the sender also receives a copy of the message. This property, called *loopback*, can be toggled by setting the `loopBack` property of the socket using `dp_socketOption`. `LoopBack` is a boolean valued property (`yes` or `no`).

- **Time to Live (TTL)**: When an IP-multicast message is sent, one of the fields in the packet is called the time-to-live, or ttl, field. When the packet is routed through a gateway, the ttl field is decremented. If the ttl field is ever zero, the gateway drops the packet. The ttl field thus limits the lifetime of a packet, preventing packets from forever wandering through the network.

  The default value for `ttl` is specified when the socket is created and can be changed later by setting the ttl property on the socket. For example, the following fragment sets the `ttl` of socket to 128:

  ```
  dp_socketOption $socket ttl 128
  ```

  The `ttl` property can be used to limit the range of a multicast. For example, to send to only hosts on you local area network, use a `ttl` value of 0. To reach all hosts within 2 network hops, set `ttl` to 2.

- **Adding and Dropping Membership**: A single IP-multicast socket can belong to several groups at once. When an IP-multicast socket is created, an initial group and port number are specified. You can join other groups by calling `dp_socketOption` with the `addMbr` command. For example, the following three lines of code create a socket that belongs to three groups

  ```
  set socket [lindex [dp_connect -mudp 225.28.199.17 2120 16] 0]
  dp_socketOption $socket addMbr 224.2.12.187
  dp_socketOption $socket addMbr 235.102.89.5
  ```

  The same port number is used for all three addresses. You can use `dropMbr` to remove yourself from a group

  ```
  dp_socketOption $socket dropMbr 235.102.89.5
  ```

The Domain Name Service

One use of the `dp_address` command is to specify the source or destination of addresses for UDP and IP-multicast sockets. Another use is to find out information about hosts and services using the `dp_address info` command. For example, suppose you create the following address:

```
A% set a [dp_address create fw.cs.cornell.edu 0]
addr1
```

You can find information about that address using the `dp_address info` command:

`load`, which contains the load average of each machine in the pool. The multicast group address and port number, 225.28.199.17 and 2120, respectively, were chosen arbitrarily.

Given this structure, you can find the most lightly loaded machine by searching through the `load` array. Alternatively, you could use Tk to build an interface that graphically displayed the contents of this array.

Note that, in the example above, each machine also receives load average reports from itself. For some applications this behavior, which is called *loopback*, may be undesirable. Loopback can be turned off using the `loopBack` socket option.

IP-multicast sockets and UDP sockets can be used to send data to each other. That is, you can use a UDP socket to send to a multicast address, and a multicast socket to send to a UDP socket. For example, the following code uses a UDP socket to report the load average to the multicast group (assuming the procedure `SendReport` is defined as above).

```
set udp [lindex [dp_connect -udp 0] 0]
set address [dp_address create 225.28.199.17 2120]
SendReport $udp $address
```

IP-multicast Socket options

IP-multicast sockets have the same socket options as UDP socket, and four other properties that

---

```
proc GetLoad {} {
   set info [split [exec uptime] ,]
   lindex [lindex $info 3] 2
}

proc SendReport {socket address} {
   set msg "[dp_hostname] [GetLoad]"
   dp_sendTo $socket $msg $address
   dp_after 1000 SendReport $socket $address
}

proc RecvReport {mode socket} {
   global load
   set info [dp_receiveFrom $socket]
   set x [lindex $info 1]
   set hostName [lindex $x 0]
   set hostLoad [lindex $x 1]
   set load($hostName) $hostLoad
}

set socket [lindex [dp_connect -mudp 225.28.199.17 2120 16] 0]
set address [dp_address create 225.28.199.17 2120]
dp_filehandler $socket r RecvReport
SendReport $socket $address
```

Figure 13: A load monitor using IP-multicast

```
B% dp_socketOption $udpB sendBuffer
9000
B% dp_sendTo $udpB [format %12000d 10] $dest
error writing file3: Message too long
B% dp_socketOption $udpB sendBuffer 15000
15000
B% dp_sendTo $udpB [format %12000d 10] $dest
B%
```

If the receive buffer is too small, the message will be dropped silently.

As with TCP sockets, each system imposes certain restrictions on the maximum size of the buffers.

- **Blocking Behavior**: `Dp_receiveFrom` normally blocks if there is no data waiting to be read. This behavior can be changed by using the `noblock` property. `Noblock` is a boolean valued property (`yes` or `no`). `Dp_sendto` never blocks.

IP-multicast

IP-multicast sockets are similar to UDP sockets, in that they transmit whole messages unreliably using `dp_sendTo`, `dp_receiveFrom`, and `dp_address`, but they have the advantage that they can efficiently send data to several clients with one function call and a single address.

To use IP-multicast, you must first understand the concept of a *group address*. A group address looks like an ordinary IP address, except the range is 224.0.0.1 to 239.255.255.255 and it is not associated with a single machine, but with a group of machines. When a process sends a message to a group address, all machines that have created a multicast socket with that group address will receive it[8]. A machine becomes part of a group by creating a socket by calling `dp_connect` with the `-mudp` flag, passing in three additional parameters: the group address, the port number, and the *time-to-live* (ttl). We will explain the meaning of the ttl parameter shortly.

For example, suppose you have a pool of Unix machines on your network that can be used for general purpose computing. You decide to use Tcl-DP to build a load monitor that reports the load on each machine in the pool to every other machine in the pool once a second so that you can find an unloaded machine. The code in figure 13 shows an implementation of this service that uses IP-multicast to send the load average of each machine to every other machine. The procedure `GetLoad` returns the load average on each machine, obtained from parsing the results of the Unix `uptime` command. The procedure `SendReport` sends a message containing the hostname and load average on the local machine to every machine in the group, and then schedules another call to `SendReport` using the `dp_after` command, which is equivalent to Tk's `after` command. `SendReport` takes two parameters, an IP-multicast socket and a group address. The `ReceiveReport` procedure is called by a file handler whenever the IP-multicast socket becomes readable (i.e., when a report has been received). `ReceiveReport` updates the global array

8. Unless, of course, the message is lost in the network. In this case, only some of the machines will receive the message.

The last parameter is created using the `dp_address` command.

`Dp_address` creates, deletes, and queries *addresses*. To create an address, you must specify the host address and a port number. For example, assuming process A is running on `mayo.sandwich.com`, the following command creates an address for the socket on port 2020:

```
B% set dest [dp_address create mayo.sandwich.com 2020]
addr0
```

The return value of `dp_address create` can then be used as a parameter to `dp_sendTo`, as shown below:

```
B% dp_sendTo $udpB “Hello there” $dest
```

Process A can read the message using `dp_receiveFrom`:

```
A% set x [dp_receiveFrom $udpA]
addr0 {Hello there}
```

`Dp_receiveFrom` returns a list of two values. The first is the address of the sender and the second is the message. The address can be used for replies:

```
A% dp_sendTo $udpA “Pleased to meet you” [lindex $x 0]
```

An important feature of UDP sockets is that they are *connectionless*. That is, a pair of TCP sockets is needed for each pair of processes that communicate. The two sockets are *connected*. In contrast, a single UDP socket can be used to communicate with an unlimited number of other processes, since the destination address is specified in the message. Since many operating systems place rather stringent limits on the number of open sockets in a process, but almost no limit on the number of addresses that can be created, UDP sockets are useful in applications that communicate with many other processes.

The standard functions `close`, `dp_atclose`, `dp_isready`, `dp_filehandler`, and `dp_socketOption` can be used with UDP sockets. The following properties can be set on UDP sockets using `dp_socketOption`:

- **Buffer Sizes**: Setting the buffer size of a UDP sockets determines the maximum message size. For example, if the send or receive buffer is 8 KBytes and you try to send a 12 KByte message, the message will not get through and you will get an error message:

Table 3: TCP Socket properties set by the `dp_socketOption` command

property	legal values	default	Description
`sendBuffer`	1-64K[a]	> 8192	Size of TCP send buffer
`recvBuffer`	1-64K	> 8192	Size of TCP receive buffer
`noblock`	yes, no	no	Will calls on the socket will block?
`autoClose`	yes, no	yes	Will the socket will automatically close and remove file handlers if the connection is broken?
`linger`	>= 0	0	Blocking time on `close` to ensure data delivery.
`reuseAddr`	yes, no	yes	Allow local address reuse?

a. The range varies from system to system, as does the default. This is a typical value.

UDP

TCP sockets provide reliable, in-order data delivery with a stream interface. In contrast, UDP sockets provide no guarantees on whether data will get through but preserves message boundaries. That is, when a message is sent by one application to another using UDP, either the entire message will get through and the receiver will receive it in as a single unit, or the message will not get through at all. There is a chance that a message may be duplicated using UDP, but such duplication is rare.

To create a UDP socket using Tcl-DP, you call `dp_connect` with the `-udp` flag and port number of the socket. For example, the following command creates a UDP socket with port number 2020 in process A:

```
A% set info [dp_connect -udp 2020]
file3 2020
A% set udpA [lindex $info 0]
file3
```

As with TCP sockets, an identifier for the socket (`file3`) and the port number of the socket (`2020`) are returned. If you can pass in a port number of `0`, the system will chose (and return) an unused port number.

For the examples that follow, we will need another UDP socket in process B:

```
B% set info [dp_connect -udp 4100]
file3 4100
B% set udpB [lindex $info 0]
file3
```

You can use the `dp_sendTo` command to send a message using UDP. `Dp_sendTo` takes three parameters: the socket identifier (`file3`), the message (“`hello`”), and the destination address.

```
A% puts “Send buffer size: [dp_socketOption $s1 sendBuffer]”
Send buffer size: 8192
A% puts “Receive buffer size: [dp_socketOption $s1 recvBuffer]”
Receive buffer size: 8192
A% dp_socketOption $s1 sendBuffer 32768
32768
A% dp_socketOption $s1 recvBuffer 32768
32768
```

Each operating system imposes certain restrictions on the maximum size of the buffers (it’s rarely greater than 64 Kbytes). In some cases, when and how you can resize them is also restricted. The best way to figure it out for your personal configuration is by experimentation.

- **Blocking Behavior**: By default, calls that write data to a socket will block if there is not enough room left in the send buffers, and calls that read data from a socket will block if there is no data waiting to be read. These behaviors can be changed by using the `noblock` property. `Noblock` is a boolean valued property (`yes` or `no`).

  The first line below queries the current value of the `noblock` property. The second line makes the socket non-blocking, so that subsequent calls to `dp_receive` do not block.

  ```
  A% dp_socketOption $s1 noblock
  no
  A% dp_socketOption $s1 noblock yes
  A% dp_receive $s1
  ```

- **Automatic socket cleanup**: By default, `dp_send` and `dp_receive` automatically close a socket and removes its file handlers when the connection is broken. If an application programmer wants to close the file manually, this behavior can be suppressed by setting the `autoClose` property of the socket. `AutoClose` is a boolean valued property (`yes` or `no`).

- **Reusing Port Numbers**: When a process with an open socket crashes, the operating system prevents other processes from opening a socket with the same port number until enough time has passed that any old packets floating around in the network that are destined for the dead process have expired. If the boolean property `reuseAddr` is set to `yes`, processes can reuse the port number in question immediately.

- **Buffer Sizes**: When a process writes data to a socket, the operating system copies that data into an internal buffer, called the *send buffer*, where it remains until its transfer is acknowledged by the receiver. As long as there is sufficient free space in the send buffer, calls like `dp_send` that write data to the socket will not block. Similarly, the receiver's operating systems has an internal buffer (the *receive buffer*) where it stores incoming data until the client executes a call such as `dp_receive` to read it.

  You can set or query the size of the send and receive buffers for each socket using the `sendBuffer` and `recvBuffer` properties. For example, the following code prints out the current size of the send and receive buffers and then sets them both to 32 KBytes

```
1     dp_MakeRPCServer 1905
2     proc SendFile {host port filename} {
3            set inFile [open $filename r]
4            set info [dp_connect $host $port]
5            set socket [lindex $info 0]
6            while {![eof $inFile]} {
7                  set data [read $inFile 8192]
8                  dp_send $socket $data nonewline
9            }
10           close $inFile
11           close $socket
12    }
```

Figure 11: Server code for Tcl-DP FTP example

```
1     proc Connect {serverHost} {
2            return [dp_MakeRPCClient $serverHost 1905]
3     }
4
5     proc GetFile {server remoteFilename localFilename} {
6            set outFile [open $localFilename w]
7            set cInfo [dp_connect -server 0]
8            set cSocket [lindex $cInfo 0]
9            set cPort [lindex $cInfo 1]
10           dp_RDO $server SendFile [dp_hostname] $cPort $remoteFilename
11           set dInfo [dp_accept $cSocket]
12           close $cSocket
13           set dSocket [lindex $dInfo 0]
14           while {1} {
15                 if [catch {dp_receive $dSocket} data] {
16                       break;
17                 }
18                 puts -nonewline $outFile $data
19                 puts -nonewline "#"
20                 flush stdout
21           }
22           puts ""
23           close $outFile
24    }
```

Figure 12: Client code for Tcl-DP FTP example

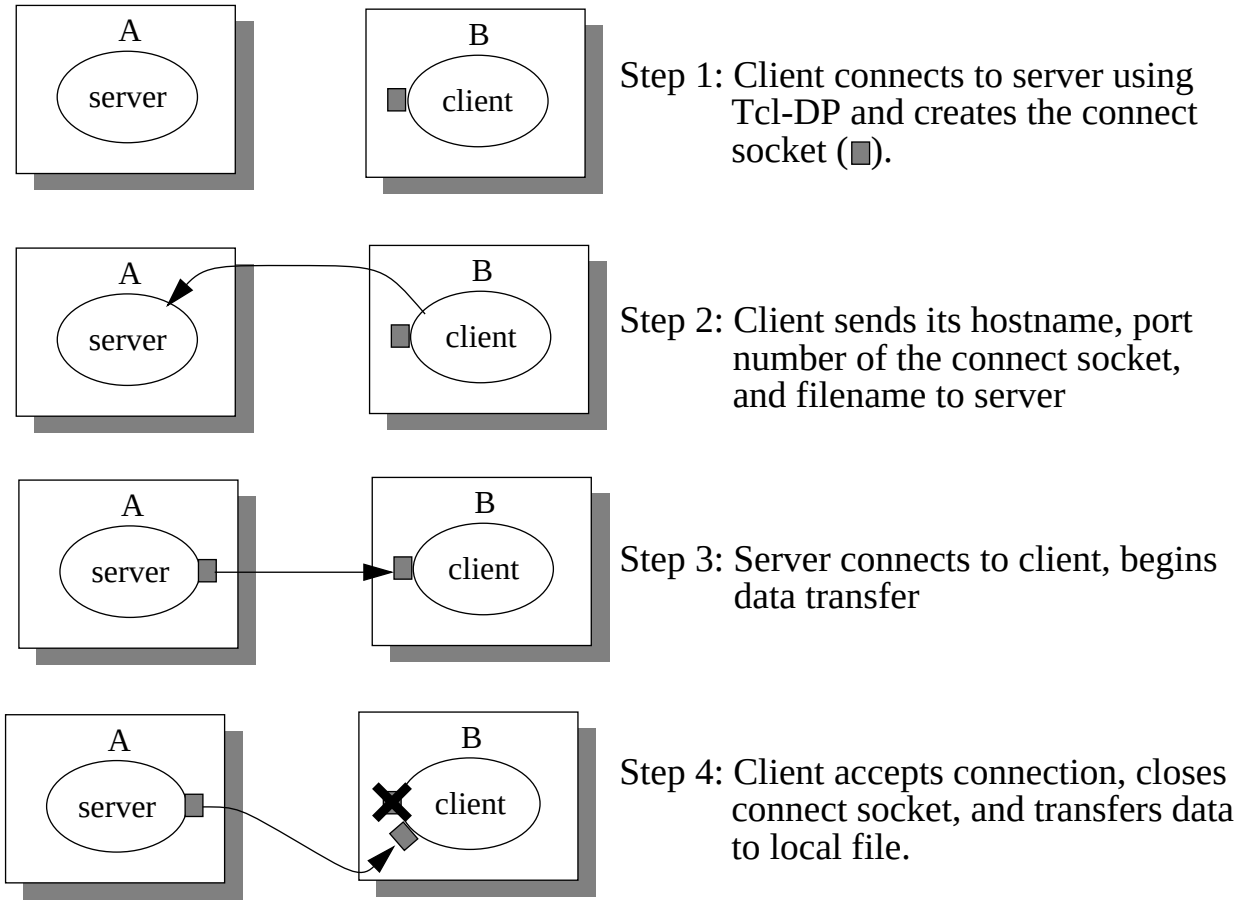

Figure 10: Mechanics of an FTP-style file transfer using Tcl-DP

The Tcl-DP code in figures 11 and 12 implement this protocol. The client uses `dp_RPC` to implement step 2 by calling the `SendFile` function in the server.

TCP Socket options

A socket has many parameters that affect its behavior. Tcl-DP sets these parameters to reasonable default values when the societe is created. *Socket options* give you control over these options. They can be used to specify whether function calls block, how data is buffered, and the reliability of the connection. They are accessed using `dp_socketOption`, which takes two or three parameters, similar to `configure` requests on Tk widgets. The first parameter is the socket identifier (e.g., `file4`), the second is a *property* of the socket you want to examine or modify, and the third parameter, if present, is the new value for the property. If the third parameter is not supplied, the current value of the property is returned. The paragraphs below discuss the properties relevant to TCP sockets, which are summarized in table 3.

7. Of course, the connection could be lost for other reasons, such as the server crashing. A better implementation of the client and server would handle this case but complicate the example.

If ‘w’ is used for this parameter, the function will be called when the socket becomes writable.

If an error occurs when a file handler callback executes, the file handler is automatically removed to prevent the program from going into an infinite loop if the file handler does not consume the data at the socket. You can manually remove a file handler by calling `dp_filehandler` without the `mode` or `callback` parameters.

Another way of detecting if a file is readable or writable is to use the `dp_isready` command. `Dp_isready` takes a file handle as a parameter and returns a list of two boolean values (`0` or `1`). The first element of the return value indicates whether the file is readable. If it is `0`, then any call that attempts to read data from the file (e.g., `gets` or `dp_receive`) will block. The second element of the return value indicates whether the file is writable. If it is `0`, then any call that attempts to write data to the file (e.g., `puts` or `dp_send`) will block.

For example, if a call to read data on `s1` would block, `dp_isready` returns `0 1`:

```
A% dp_isready $s1
0 1
```

If B sends some data to A:

```
B% dp_send $s2 “hi there”
```

a call to read data on `s1` would not block, so `dp_isready` returns `1 1`:

```
A% dp_isready $s1
1 1
A% dp_receive $s1
hi there
```

Example: A Simple FTP Server

Suppose you wanted to implement a simple FTP style server using Tcl-DP. One way to implement it is to create a new TCP connection for each file transfer, which would be a four step process, as illustrated in figure 10.

1. The client opens the output file and creates a listening socket (the *connect socket*)
2. The client sends the following information to the server: the client’s hostname, the port number of the connect socket, and the filename to transfer.
3. The server opens the input file and a TCP connection to the client’s connect socket, which gives the server a *data socket*. It then enters a loop where it repeatedly reads the input file and sends the data over to the client. It then closes the data socket and input file.
4. Meanwhile, the client accepts the connection on the connect socket, closes it, and enters a loop where it receives data from the server and write it to the output file. If `dp_receive` ever returns an error, it means the connection was broken, presumably because the transfer is complete[7]. Since `dp_receive` automatically closes the file when the connection is broken, the client only has to close the output file before returning.

```
A% dp_packetReceive $s1
message 1
A% dp_packetReceive $s1
message 2
```

It is possible that only part of the message is available at the time `dp_packetReceive` is called. In this case, `dp_packetReceive` will buffer the partial result internally and return an empty string. A subsequent call to `dp_packetReceive` will return the entire packet.

To preserve message boundaries, `dp_packetSend` attaches a binary header onto the message, which `dp_packetReceive` strips. The presence of this header means that applications must be careful about intermixing calls to `dp_packetSend` and `dp_packetReceive` with `dp_send` and `dp_receive`, and other data transmission functions.

File handlers

So far, the socket functions we have seen block if no data is present on the socket. That is, the function call will not return until some data arrives. Blocking can cause problems, for example, if a program needs to read data from several connections at once. To address this problem, Tcl-DP provides a mechanism called *file handlers* that arranges for a Tcl function to be called whenever the file becomes *readable* or *writable*. A socket becomes readable when another process attempts to connect to it, in the case of a listening socket, or when it has data waiting at its input, in the case of a data socket. A data socket become writable whenever a call to `dp_send`, puts, or `dp_packetSend` will not block.[6]

File handler callback procedures take two parameters. The first parameter, called the *mode*, indicates whether the socket has become readable or writable. It will be '`r`' if the socket is readable, or '`w`' if the socket is writable. The second parameter is the handle of the socket. For example, the following fragment arranges for A to accept a new connection whenever one is requested on its listening socket. The procedure `MyAccept` calls `dp_accept` to accept the connection, prints a message on the screen, and adds the new socket to the `socketList` variable.

```
A% proc MyAccept {mode file} {
      global socketList
      set info [dp_accept $file]
      set newSocket [lindex $info 0]
      puts "Accepted connection from [lindex $info 1]"
      lappend socketList $newSocket
   }
A% dp_filehandler $listeningSocket r MyAccept
```

The call to `dp_filehandler` arranges for `MyAccept` to be called whenever `listeningSocket` becomes readable. The second parameter to `dp_filehandler` ('`r`' in the example above) indicates that the file handler should only be called when the socket is readable.

---

6. Such calls can block if they are communicating over a particularly slow connection, since the system will only buffer a limited amount of data. The amount of data buffered can be adjusted using the `dp_socketOption` command, discussed below.

Sending and receiving data

The simplest way to send data from one application to another is to use the Tcl functions `gets`, `read`, and `puts`. For example, in the following fragment B sends the string "Hello world" to A:

```
B% puts $s2 "hello world"
```

To receive the string, A calls `gets`:

```
A% gets $s1
hello
```

Another interface for sending and receiving data is `dp_send` and `dp_receive`. `Dp_send` takes the same arguments as `puts` and serves the same function. `Dp_receive` is similar to the Tcl `read` command, except it takes an optional `-peek` flag indicating that the data should be read from the socket, but not consumed, so that a subsequent call to `dp_receive` will see the same data. In addition, if the connection is ever broken, `dp_send` and `dp_receive` automatically close the socket.

TCP sockets provide a stream interface, which can cause unexpected results if you want to use them to send messages between processes. For example, suppose B sends several messages to A. When A reads its socket, the messages might be concatenated or only part of a message may be present. The following code fragment shows this effect in action. If B executes the following commands

```
B% dp_send $s2 "message 1"
B% dp_send $s2 "message 2"
```

When A calls `dp_receive`, it gets both messages at once:

```
A% dp_receive $s1
message 1
message 2
```

Since some applications want to preserve message boundaries and want the reliability of TCP, Tcl-DP provides two functions, `dp_packetSend` and `dp_packetReceive`, that provide message-oriented delivery. For example, suppose B uses `dp_packetSend` instead of `dp_send` in the example above:

```
B% dp_packetSend $s2 "message 1"
B% dp_packetSend $s2 "message 2"
```

When A calls `dp_packetReceive`, it gets one message per function call.

After creating the listening socket, the server typically waits for a connection to arrive by calling `dp_accept`, which will return when another process attempts to connect to the socket. For example, the following code causes A to block while waiting for a client:

```
A% set newClient [dp_accept $listeningSocket]
```

Another process connects to A using another form of `dp_connect`. In this form, the hostname of the machine on which the server is running, and the port number of the server socket, are passed as parameters to `dp_connect`. If the hostname of A is `mayo.sandwich.com`, the following code will connect machine B to A:

```
B% set info [dp_connect mayo.sandwich.com 1905]
file4 3833
B% set s2 [lindex $info 0]
```

As with the previous call to `dp_connect`, a handle to the socket (e.g., `file4`) and the operating system selected port number of the socket (3833) are returned.

B's attempt to connect to A will cause A's call to `dp_accept` to return, setting the `newClient` variable to the handle of the new socket (e.g., `file5`) and the Internet address of the connecting process (e.g. `128.83.218.21`)

```
A% set newClient [dp_accept $listeningSocket]
file5 128.83.218.21
A% set s1 [lindex $newClient 0]
file5
```

Unix Domain Sockets

The example above created *Internet domain* sockets. That is, B uses an Internet address and port number to rendezvous with A's socket. On Unix systems, another naming scheme, called *Unix domain* sockets, can be used to available for connecting processes if they reside on the same machine. In this case, a file name (e.g., `/tmp/mysocket`) is used to name the socket. The following example shows a connection using Unix domain sockets.

Server code:

```
% set f [dp_connect -server /tmp/mysocket]
file4
% set s1 [dp_accept $f]
```

Client Code:

```
% set s2 [dp_connect /tmp/mysocket]
file4
```

Regardless of whether Unix or Internet domain sockets are used, processes communicate using the handles of the sockets as arguments to Tcl-DP function. In the examples, these handles are stored in the variables `s1` and `s2`.

will discuss the primitives associated with unconnected (UDP and IP-multicast) sockets later.

TCP sockets

Connecting two processes using TCP is a three step process, illustrated in figure 9. First, one process, say A, creates a *listening* socket with an associated *name* so that other processes can contact it. The name can be either a Unix filename or an Internet address and port number. Second, another process (B) creates another socket and *connects* to A's listening socket. Third, A *accepts* the connection, which creates a new socket so that other processes can contact A using the A's listening socket.

In Tcl-DP, `dp_connect` is used for steps one and two, and `dp_accept` is used for step three. `Dp_connect` will create a listening sockets if the `-server` flag is provided. For example, the following command creates a listening socket on port 1905 and assigns the socket identifier (`file4`) to the variable `listeningSocket`.

```
A% set info [dp_connect -server 1905]
file4 1905
A% set listeningSocket [lindex $info 0]
file4
```

The third parameter to `dp_connect` is the port number. Only one socket can be associated with a given port at any time. If another socket is already open on that port, `dp_connect` will return an error. You can have the operating system select an unused port by specifying a port number of `0` to `dp_connect`. No matter who selects the port number, `dp_connect` will return a list of two values: the identifier for the socket and its port number.

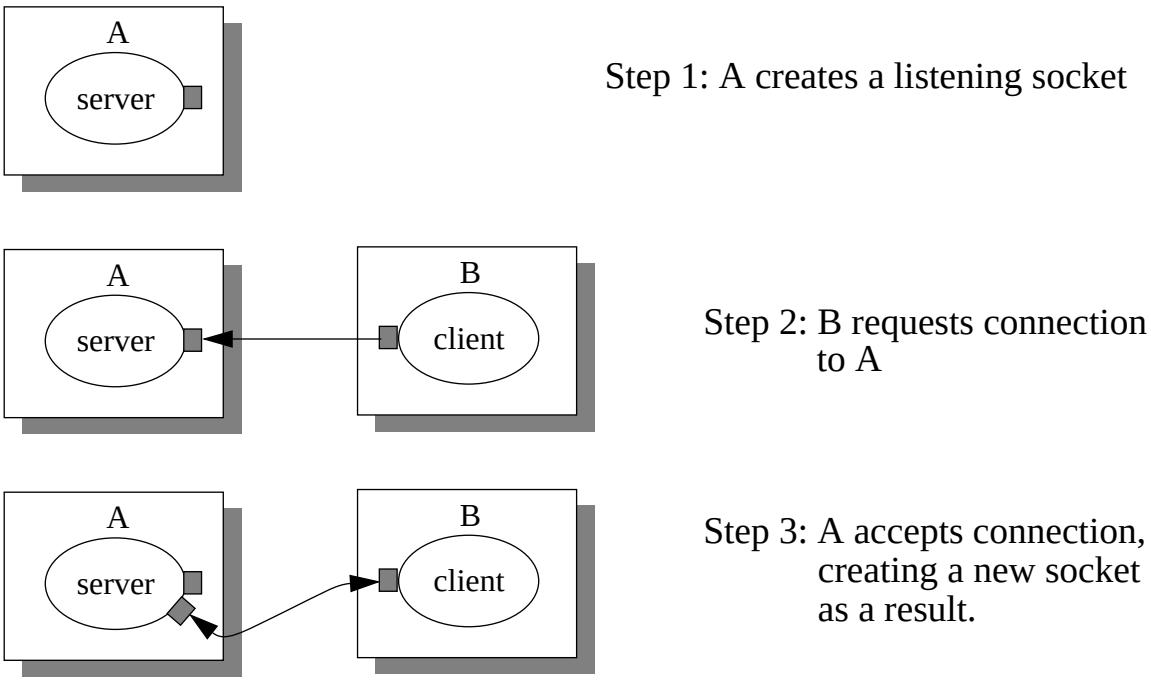

Figure 9: Connecting processes using TCP

```
ns_Register addService /demo/whiteboard mayo.sandwich.com \
            “dptcl -f /home/tcldp/ns/wbServer.tcl”
ns_Register aliasService /demo/whiteboard /wbServer
```

The name server can associate several names, called aliases, with a single process. A call to `NS_GetServiceConn` will search through service names and their aliases for a match. The name server supports pattern matching on service names similar to Unix file name matching to locate servers. The interface to this pattern matching function is `NS_FindService`. `NS_FindService` returns a list of all the matching names, similar to the way the Tcl `glob` command returns a list of file names that match a pattern. For example, the following call locates all servers in the demo tree:

```
B% set demoServers [NS_FindService /demo/*]
```

The use of the slash (“/”) character to give a hierarchical structure to the process names is only a convention. `NS_FindService` uses the Tcl string match command to search for matches. Consequently, you can establish any naming scheme you like, but we encourage you to use the naming convention presented above (i.e., use slash characters) to ensure uniformity.

The name server is a network wide service built using Tcl-DP. It runs on a well known host and port number in your network. The exact host and port are specified in the file `$dp_library/ns/nsconfig.tcl`. This immediately bring to mind the following questions: who starts the name server and what happens if the machine that runs the name server crashes? The answer to the first question is that the name server is typically started by an entry in `/etc/rc.local` when a designated machine boots. The answer to the second question is that backup copies of the name server can be run at the same time as the primary server. The machines on which the backup server are run are specified in the file `$dp_library/ns/nsconfig.tcl`. All name server functions accessible from the client, such as `NS_GetServiceConn` and `NS_FindService`, will locate the primary server and connect to it. If the primary name server crashes, the backups elect a new primary name server. More details on the design and implementation of the name server are available elsewhere [tcl-95 ref].

Tcl-DP Communication Services

This section shows you how to use Tcl-DP’s interfaces to TCP, UDP, and IP-multicast. It also shows you how to query and set various properties of sockets, such as buffer sizes and blocking properties, to gain more control over the properties of the communication channel. Finally, this section will show you how to access the Internet Domain Name Service (DNS) using Tcl-DP. The DNS maps internet addresses (e.g., 128.32.149.117) to host names (e.g., toe.cs.berkeley.edu).

Review of Berkeley Sockets

Before discussing Tcl-DP’s mechanisms for connecting two processes, we will briefly review Berkeley sockets. Sockets come in several varieties, distinguished by the communication protocol (e.g., TCP or UDP), how the socket is identified, and whether or not the socket is the connection initiator. In this section, we will discuss the primitives used for *connected* (i.e., TCP) sockets. We

```
A% proc Subscribe {} {
      global dp_rpcFile clients log
      lappend clients $dp_rpcFile
      dp_atclose $dp_rpcFile append \
         "set clients [ldelete $dp_rpcFile $clients]"
      foreach cmd $log {
         eval dp_RDO $dp_rpcFile $cmd
      }
   }
```

The Name Server

As distributed applications get more complex, starting and stopping servers, locating a server and the port on which it is listening, and keeping track of the server state becomes more of a problem. For example, in our whiteboard program, we want to start the client application and have it connect to a running server if one is available or start one if it is not running. The problem is how do we locate a running server or start one if it is not running? The Tcl-DP *name server* solves this problem.

The name server associates a name with each process which are patterned after Unix file names. For example, the name of the whiteboard server might be `/demo/whiteboard`.

When the server starts up, it uses the procedure `NS_SrvcInit` to contact the name server and declare its name[5]. For example, the following commands tells the name server that the whiteboard server is running on `mayo.sandwich.com`, port 4500:

```
A% NS_SrvcInit /demo/whiteboard mayo.sandwich.com 4500
```

Registering a name with the name server adds it to the list of server that the name server knows about. We call a registered name a *service*. When a whiteboard client wants to locate the whiteboard server, it issues an `NS_GetServiceConn` call to find the service. `NS_GetServiceConn` takes the name of a service as an argument, contacts the name server, and returns a host and port number where the server can be contacted. For example, the code

```
B% set whiteboardServer [NS_GetServiceConn /demo/whiteboard]
```

queries the name server to get the host and port number of the whiteboard server which can be passed to `dp_MakeRPCClient`.

If the server is not running when `NS_GetServiceConn` is called, the name server can start the process for you if the service is registered as an *autostart* service. Only processes marked as autostart in the file `$dp_library/ns/nsconfig` can be started automatically. For example, adding the following line to this file makes the whiteboard an autostart service:

---

5. The prefix for all name server commands is `NS_`

allow inbound RPC's to be checked on any socket (client side or server side), use the `dp_SetCheckCmd` function. For example, to add client side command checking, the client can execute the following command:

```
dp_SetCheckCmd $server ClientCheckCommand
```

where `ClientCheckCommand` is a command checking procedure similar to `WhiteboardCmdCheck` in figure 8. Alternatively, the client check command can be specified when the connection is made:

```
set server [dp_MakeRPCClient $host 4544 ClientCheckCommand]
```

Cleanup

In distributed programs like the whiteboard example, clients and servers crash or shutdown without warning. These crashes can cause unexpected, often fatal, errors to occur. For example, if a whiteboard client dies unexpectedly, the server wants to remove the client from the `clients` variable. The `dp_atclose` command is designed to handle such clean up actions automatically.

`Dp_atclose` associates a list of Tcl commands with a file. Just before the file is closed, which happens automatically if a connection is broken, each command in the list is called. The first argument to `dp_atclose` is a file identifier (e.g., `$server`) that specifies the target file or connection and the second argument is a command. Valid commands are: `append`, `remove`, `appendUnique`, `insert`, and `list`. `Append` adds a new callback to the end of the list. `Remove` deletes a previous appended callback. `AppendUnique` adds a callback to the end of the list, but only if it is not already part of the list. `Insert` places a callback at the beginning of the list, and `list` returns the current callback list. Table 2 lists the valid commands and arguments for `dp_atclose`[4].

Table 2: Arguments for `dp_atclose` and `dp_atexit`

Command	Arguments	Description
`append`	`callback`	Invoke `callback` when file closes
`appendUnique`	`callback`	Invoke `callback` precisely once when file closes
`list`	`-`	Return file closing callback list
`remove`	`callback`	Remove `callback` from file closing callback list
`insert`	`callback`	Insert `callback` at beginning of callback list

We can use `dp_atclose` in the whiteboard program to remove a client that has crashed from the `clients` variable by making the following change to the `Subscribe` command. The modified code is shown in boldface:

4. Tcl-DP has another cleanup command, `dp_atexit`, that is similar to `dp_atclose`. `Dp_atexit` callbacks execute just before the program exits.

other commands are disallowed.

```
A% proc CheckCmd {cmd args} {
      case $cmd in {
         Subscribe return;
         puts{
            set file [lindex $args 0]
            if {[string compare $file stderr] != 0]} {
               return -code break
            }
            return;
            }
         set{
            if {[llength $args] != 1} {
               return -code break
            }
            return;
            }
         eval {return -code continue}
         catch {return -code continue}
         if {return -code continue}
      }
      return -code break;
   }
```

We will now use both features to make the whiteboard program more secure. We will only allow clients whose IP-address is in the `whiteboard-clients` file to connect, and we will verify that the clients are executing legal commands. The modified code is shown in figure 8.

Of course, the server can still execute commands in the client. This capability can cause problems in environments where the user can not verify that a server is authentic. For example, a client can innocently connect to a server and the server can remove all files in the client environment. To

```
1     # Set the list of allowed clients from whiteboard-clients
2     set f [open whiteboard-clients r]
3     dp_host -
4     while {[get $f host] != -1} {
5           dp_host +$host
6     }
7     close $f
8
9     # The only allowed commands are Subscribe and Publish
10    proc WhiteboardCmdCheck {cmd args} {
11          case $cmd in {
12                Subscribe return
13                Publish return
14          }
15          return -code break;
16    }
17
18    dp_MakeRPCServer 4545 dp_CheckHost WhiteboardCmdCheck
```

Figure 8: Extra Commands for Secure Tcl-DP Shared Whiteboard Server

```
A% proc CheckConnection {file addr} {
      if {[string match $addr 128.32.134.*] != 1]} {
         error "Host not authorized"
      }
   }
A% dp_MakeRPCServer 4545 CheckConnection
```

The default login procedure for Tcl-DP is `dp_CheckHost`, which provides a simple access control list mechanism, similar to `xhost` in the X window system, for limiting connections to a set of IP host addresses. The access control list is modified by the `dp_host` command. For example, the following Tcl-DP commands allow connections from machines in the `128.32.134` subnet except `128.32.134.117`, or from the machine named `mayo.sandwich.com`.

```
A% dp_host -
A% dp_host +128.32.134.*
A% dp_host -128.32.134.117
A% dp_host +mayo.sandwich.com
A% dp_MakeRPCServer 4567
```

By default, connections from any host are allowed (equivalent to `dp_host +`). The `dp_host` command and its associated `loginFunc` are implemented entirely in Tcl. They can be found in the file `$dp_library/acl.tcl` in the distribution. This code can be used as an example for building more complex login security functions. For example, a server could maintain a list of authorized users and passwords and require a client to explicitly login. Or, a server could use a system such as PGP or Kerberos to authenticate clients.

The login procedure can prevent rogue users from accessing a server, but even innocent users can accidentally run commands with horrible side effects. Such mistakes are particularly disastrous if the server is running as root. For example, we all want to stop someone from accidentally running

```
B% dp_RDO $server exec rm -rf /
```

To prevent such catastrophes, `dp_MakeRPCServer` takes a second optional argument, called the *check command,* which checks each command from a `dp_RPC` or `dp_RDO` call before it is run. The return code from the check command specifies whether to disallow the command, to continue checking sub commands, or to allow the command to be executed with no further checking. If the procedure returns a normal value, the command is allowed and no further checking is performed. If the option `-code break` is used with the `return`, the command is disallowed. If `-code continue` is used, the command is allowed but nested commands are checked. Notice that the nested checking allows commands such as

```
B% dp_RPC $server eval rm -rf /
```

to be caught.

To illustrate the use of check commands, the following code defines a procedure that allows `puts` to be run on standard error, `set` to be run with one argument, `Subscribe` to be run with now further checking, and `eval`, `catch`, and `if` to be run with embedded command checking. All

```
1     #!/usr/local/bin/dpwish -f
2     puts “Enter hostname of server:”
3     gets stdin host
4     set server [dp_MakeRPCClient $host 4544]
5     dp_RDO $server Subscribe
6     proc DoCmd {args} {
7         global server
8         eval dp_RDO $server Publish $args
9     }
10    wm grid . 1 1 1 1
11
12    # Create menu bar:
13    frame .menubar -relief ridge
14    menubutton .menubar.file -text “File” -menu .menubar.file.menu
15    pack .menubar.file -side left
16    menubutton .menubar.object -text “Objects” -menu .menubar.object.menu
17    pack .menubar.object -side left
18    pack .menubar -side top -fill both
19    menu .menubar.file.menu
20    .menubar.file.menu add command -label “Exit”  -command exit
21    menu .menubar.object.menu
22    .menubar.object.menu add command -label “Clear” -command “DoCmd .c delete all”
23    .menubar.object.menu add command -label “Circle” -command “DoCmd CreateCircle”
24
25    # Create canvas, procs, bindings
26    canvas .c -background green
27    pack .c -fill both
28
29    proc CreateRect {x y} {
30          DoCmd .c create rectangle $x $y $x $y -width 4 -outline white
31    }
32    proc CreateCircle {} {
33          set i [.c create oval 150 150 170 170 -fill skyblue]
34          .c bind $i <Any-Enter> “DoCmd .c itemconfig $i -fill red”
35          .c bind $i <Any-Leave> “DoCmd .c itemconfig $i -fill SkyBlue2”
36          .c bind $i <2> “DoCmd plotDown .c $i %x %y”
37          .c bind $i <B2-Motion> “DoCmd plotMove .c $i %x %y”
38    }
39    proc Clear {} {DoCmd .c delete all}
40    proc plotDown {w item x y} {
41          global plot
42          $w raise $item
43          set plot(lastX) $x
44          set plot(lastY) $y
45    }
46    proc plotMove {w item x y} {
47          global plot
48          $w move $item [expr $x-$plot(lastX)] [expr $y-$plot(lastY)]
49          set plot(lastX) $x
50          set plot(lastY) $y
51    }
52
53    bind .c <B1-Motion> {CreateRect %x %y}
```

Figure 7: Tcl-DP Shared Whiteboard Client

connect the client to the server. The `DoCmd` procedure defined in lines 6-9 uses `dp_RDO` to call `Publish` in the server, which sends whiteboard commands to the clients. The `CreateRect`, `CreateCircle`, and `Clear` routines use `DoCmd`.

Security

One problem with this server is that any client can connect, and a connected client can execute any command. Tcl-DP uses two mechanisms to handle these two different security holes.

The first level of defense Tcl-DP provides is an optional "login" procedure that can be supplied with the `dp_MakeRPCServer` command,. This procedure allow a server to specify a Tcl procedure that will be executed when a client connects to the server. The procedure is called with the file handle and IP address of the new client (e.g., `file4` and `128.32.133.117`) as arguments. For example, the following server logs all connection requests to a file.

```
A% set logFile [open /tmp/connect.log w]
A% proc LogConnection {file addr} {
       global logFile
       puts $logFile "Connection accepted from $addr on $file"
   }
A% dp_MakeRPCServer 4545 LogConnection
```

The login procedure can be used to prevent illegal connections. If the connection is determined illegal, the login procedure should return an error. For example, the following server only allows connections from hosts in the `128.32.134` subnet.

```
1     #!/usr/local/bin/dpwish -f
2     dp_MakeRPCServer 4544
3
4     set clients {}
5     set log {}
6
7     proc Subscribe {} {
8         global dp_rpcFile clients log
9         lappend clients $dp_rpcFile
10        foreach cmd $log {
11           eval dp_RDO $dp_rpcFile $cmd
12        }
13    }
14
15    proc Publish {args} {
16        global clients log
17        lappend log $args
18        foreach i $clients {
19           eval "dp_RDO $i $args"
20        }
21    }
```

Figure 6: Tcl-DP Shared Whiteboard Server

```
1     #!/usr/local/bin/wish -f
2     wm grid . 1 1 1 1
3
4     # Create menu bar:
5     frame .menubar -relief ridge
6     menubutton .menubar.file -text “File” -menu .menubar.file.menu
7     pack .menubar.file -side left
8     menubutton .menubar.object -text “Objects” -menu .menubar.object.menu
9     pack .menubar.object -side left
10    pack .menubar -side top -fill both
11    menu .menubar.file.menu
12    .menubar.file.menu add command -label “Exit”  -command exit
13    menu .menubar.object.menu
14    .menubar.object.menu add command -label “Clear” -command “.c delete all”
15    .menubar.object.menu add command -label “Circle” -command “CreateCircle”
16
17    # Create canvas, procs, bindings
18    canvas .c -background green
19    pack .c -fill both
20
21    proc CreateRect {x y} {
22        .c create rectangle $x $y $x $y -width 4 -outline white
23    }
24    proc CreateCircle {} {
25        set i [.c create oval 150 150 170 170 -fill skyblue]
26        .c bind $i <Any-Enter> “.c itemconfig $i -fill red”
27        .c bind $i <Any-Leave> “.c itemconfig $i -fill SkyBlue2”
28        .c bind $i <2> “PlotDown .c $i %x %y”
29        .c bind $i <B2-Motion> “PlotMove .c $i %x %y”
30    }
31    proc Clear {} {.c delete all}
32    proc PlotDown {w item x y} {
33        global plot
34        $w raise $item
35        set plot(lastX) $x
36        set plot(lastY) $y
37    }
38    proc PlotMove {w item x y} {
39        global plot
40        $w move $item [expr $x-$plot(lastX)] [expr $y-$plot(lastY)]
41        set plot(lastX) $x
42        set plot(lastY) $y
43    }
44
45    bind .c <B1-Motion> {CreateRect %x %y}
```

Figure 4: A Simple Tcl/Tk Whiteboard

to the whiteboard by executing `dp_MakeRPCClient` and calling the `Subscribe` procedure. The server maintains a list of all clients connected in the global variable `clients` and a history of all whiteboard commands in the global variable `log`. When a new client is added, the commands in the `log` are sent to the new client so that it’s display is brought up to date with the other clients. The command `Publish` is called when a client executes a whiteboard command. It writes the command to the `log` and broadcasts the command to all the clients.

The modified client code is shown in figure 7, with the modified code in boldface. Lines 2-5

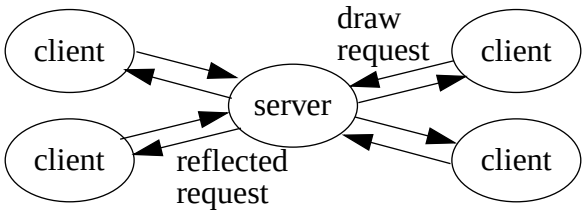

Figure 5: Architecture of the shared whiteboard example

to a new position by pressing the middle button down while the mouse is over the circle and moving the mouse. To clear the whiteboard, select the `Clear` menu item in the `Object` menu.

This code can be adapted to a shared whiteboard by broadcasting every change to the canvas, whether through bindings or procedure calls, to the other whiteboards. To handle the broadcasts, we will use a centralized server process as a reflector. Each client connects and subscribes to the whiteboard. The clients and server form a “star” with the server at the center as shown in figure 5. When a client wants to execute a whiteboard command, it sends the command to the server, which broadcasts the command to all the clients, where they are executed.

The Tcl-DP code to create the server is shown in figure 6. The call to `dp_MakeRPCServer` on line 2 initializes the server and listens for connections from clients on port 4544. A client connect

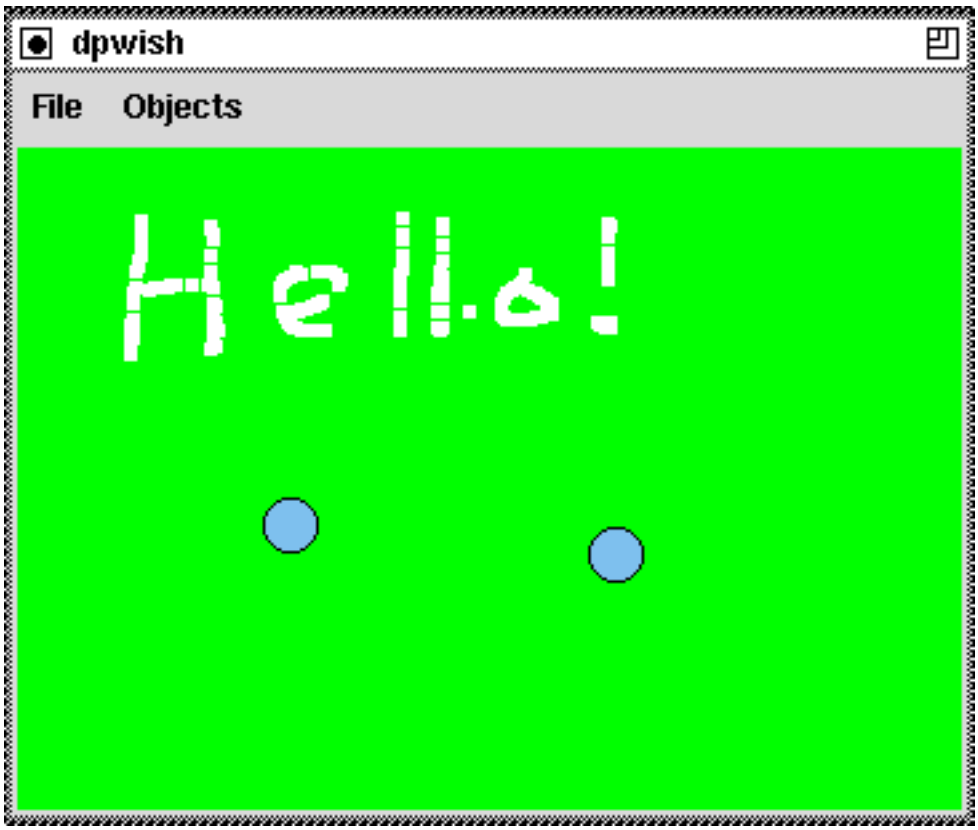

Figure 3: A simple whiteboard

```
B% proc Setup {} {
      global server scode
      dp_RDO $server -callback {set scode} Subscribe
      dp_waitvariable scode
      dp_RDO $server puts $scode
   }
B% Setup
```

Dp_waitvariable calls Tk_DoOneEvent repeatedly until scode changes value, which happens when the server sends back the subscription identifier.

In addition to the -callback option, dp_RDO has an -onerror option that specifies a Tcl fragment that will be evaluated if the remote procedure call terminates with an error. This option can be used, for example, to trap errors that occur in the remote execution of dp_RDO. To see -onerror in action, try the following fragment:

```
B% dp_RDO $server -onerror puts Greeting arg1 arg2
```

Since the Greeting procedure only takes one argument, this dp_RDO call will trigger an error, which will be printed on the screen using puts.

The final topic in this section describes how to close connections. Since the connection identifier is an ordinary file descriptor, the Tcl close command can be used to terminate the connection. For example, the command

```
B% close $server
```

shuts down the connection between B and A.

An Extended Example

To show how the functions described in this section are used in an application, we will show you how to build a simple distributed whiteboard. Before showing how this application works across a network, it is simplest to learn how the non-distributed version works.

The Tk code for a stand-alone whiteboard, which can be found in the file wb.tcl in the examples/whiteboard subdirectory in the Tcl-DP distribution, is shown in figure 4. This code creates a canvas and a menu bar, as shown in figure 3. The functions CreateRect and CreateCircle create rectangles and circles on the canvas. The function Clear deletes all objects on the canvas. The functions PlotDown and PlotMove work together to move a previously created circle.

The whiteboard is used as follows. To create rectangles, press the left button down in the main window and move the mouse while holding the button down. A trail of small, 4 by 4 pixel, squares will follow the mouse on the canvas. Internally, this response is implemented by binding (at line 51) the Button-1-motion event to call the CreateRect function.

To create a circle, select the Circle menu item in the Object menu. You can move the circle

procedure using dp_RDO and sets the local variable scode to the subscription code:

```
B% dp_RDO $server -callback {set scode} Subscribe
B% CreateUserInterface
```

When Subscribe completes, the Tcl fragment set scode is evaluated in B with the new identifier appended.

One problem that can arise using -callback is synchronizing the client and server. For example, suppose the client, after creating the user interface, must execute the Ready procedure on the server which takes the identifier returned from the Subscribe function as an argument.

```
B% proc Setup {} {
      global server scode
      dp_RDO $server -callback {set scode} Subscribe
      dp_RDO $server puts $scode
   }
B% Setup
can't read "scode": no such variable
```

This code fails on the second dp_RDO because the client has not processed the callback of the first dp_RDO, which sets the scode variable. This problem is called a client/server *synchronization problem*. To understand the solution to the synchronization problem, we must take a brief detour into the implementation of Tcl-DP.

Tcl-DP uses TCP sockets for dp_RDO and dp_RPC. In Unix, sockets are represented by files which are *readable* when the socket has data waiting to be read. Tk contains a mechanism, called *file handlers*, that automatically invokes a C callback function whenever a file is *readable*. The callback is issued from the Tk_DoOneEvent function, which invokes callbacks in response to X window events, file events, and timer events.

Tcl-DP uses file handlers and TCP sockets to implement the RPC mechanisms. In particular, dp_MakeRPCClient creates a socket and a file handler on the socket that reads strings that come in on the socket, evaluates them as Tcl commands, and returns the result. But the file handler associated with a Tcl-DP socket is not invoked until the client calls Tk_DoOneEvent.

The implementation of -onerror and -callback use dp_RDO. In the example above, A uses dp_RDO to set scode in B. Since the response by A is passed to B using dp_RDO, scode is not set until the client calls Tk_DoOneEvent. So, to solve the synchronization problem, we have to call Tk_DoOneEvent until the server's response is received.

Tcl-DP provides two Tcl commands to call Tk_DoOneEvent: dp_update and dp_waitvariable. Dp_update calls Tk_DoOneEvent repeatedly until no X, timer, or file events are pending. Dp_waitvariable calls Tk_DoOneEvent until a specified variable changes value. The solution to the synchronization problem above can use dp_waitvariable:

values. The `Subscribe` procedure given above is an example of such a procedure. The purpose in calling `Subscribe` is not to get a return value, in fact, it does not return a value, but rather to modify a global variable in the server. Whenever a procedure is called that does not return a useful value, it can be called with `dp_RDO` instead of `dp_RPC`. For example, the following calls `Subscribe` procedure using `dp_RDO`:

```
B% dp_RDO $server Subscribe
```

Besides preventing deadlock, `dp_RDO` is also more efficient than `dp_RPC`. The difference can be dramatic. Depending on the distance to the remote site, the load on the network, and the responsiveness of the server, `dp_RPC` can take anywhere from 2 to 200 milliseconds (or more!) to complete. In contrast, `dp_RDO` typically returns within a fraction of a millisecond. Moreover, `dp_RDO` reduces load on the network, client, and server, since the server does not send back a response, the client does not process a response, and the network does not transport the response.[3]

Another consequence of using `dp_RDO` is that the client and server can run in parallel. For example, in its initialization code, a typical client of our server will create a user interface and subscribe to the server database. A typical calling sequence might look like:

```
B% dp_RPC $server Subscribe
B% CreateUserInterface
```

By replacing the `dp_RPC` call in the second line with `dp_RDO`, the client can create the user interface while the server executes the `Subscribe` code.

`Dp_RDO` has two important options: `-callback` and `-onerror`. The `-callback` option is used when the return value from the remote procedure call is of interest to the client, but you want to use the parallelism provided by `dp_RDO`. For example, suppose we modify the `Subscribe` procedure to return a subscription code that the client uses to identify itself to the server.

```
A% set code 0
0
A% proc Subscribe {} {
       global dp_rpcFile clientList code
       if {[lsearch $clientList $dp_rpcFile] == -1} {
          lappend clientList $dp_rpcFile
       }
       incr code
       return $code
   }
```

You might think that the client can not use `dp_RDO` to call `Subscribe` because it needs to return a value to the caller. The `-callback` flag to `dp_RDO` is designed to handle this case. The argument to `-callback` is a Tcl script that is evaluated in the client with the return value from the remote call appended. For example, the following code calls the modified `Subscribe`

3. That is, assuming the application does not request a return value using the `-callback` option, described below.

```
B% dp_RPC $server -events {rpc x} Subscribe
```

Finally, to process all events, use the event type `all`:

```
B% dp_RPC $server -events all Subscribe
```

Table 1: Event types that `dp_RPC` recognizes

Event type	Meaning
`x`	Events from the X window system (created with Tk's `bind` command)
`file`	Events that occur on a file or socket (created with Tcl-DP's `dp_filehandler` command).
`rpc`	Same as the file event type
`timer`	Timer events (created with Tk's `after` command)
`idle`	Events that correspond to when-idle events (such as display updates, window layout calculations, and tasks schedules with `dp_whenidle`)
`all`	Same as the list `{x file timer idle}`
`none`	Don't process any events; block

A second way to prevent deadlock is to use the `-timeout` option to `dp_RPC`. If the `dp_RPC` call does not return within the specified timeout, which is given in milliseconds, `dp_RPC` returns with an error. Since it can trigger an error, `-timeout` is typically used in combination with Tcl's `catch` command. For example, the following code calls the `Subscribe` procedure on A, but prints a message on the screen if A does not respond within 100 milliseconds.

```
B% if [catch {dp_RPC $server -timeout 100 Subscribe}] {
      puts "Couldn't register with server"
   }
```

As an alternative to catching the error, you can use the `-timeoutReturn` option to specify a fragment of Tcl code to be executed if the `dp_RPC` call times out. The code is called with the connection id of the failed callback appended. The example above could be expressed like this

```
B% proc HandleTimeout {file} {
      puts "Couldn't register with server"
   }
B% dp_RPC $server -timeout 100 -timeoutReturn HandleTimeout Subscribe
```

The third way to prevent deadlock in Tcl-DP is to use a non-blocking RPC rather than a blocking RPC. The command `dp_RDO`, which stands for "remote do," initiates the RPC but does not wait for a response from the remote interpreter. Instead, it simply sends a message containing the request to the remote interpreter and immediately processes the next command in the script.

`Dp_RDO` is ideal for procedure calls that are used for their side effects rather than their return

We will discuss each mechanism in turn.

The first way to prevent deadlock is to force processes to respond to inbound RPC requests while waiting for a previously issued requests to return. If this feature was used in the example above, B would process the `puts "Pleased to meet you"` call from A while waiting for the `dp_RPC` call to return. Thus, A's RPC to B would return, allowing the remote call to `Greeting` to return, so that B's RPC to A would return. The key to implementing this strategy is to get B to process incoming RPC's while waiting for an outstanding RPC to return.

`Dp_RPC` will process inbound RPCs while waiting for an outbound RPC to complete if it is called with the `-events` option. In other words, if we used the following code to call `Greeting` on A, the system won't deadlock:

```
B% dp_RPC $server -events rpc Greeting "Hello there"
```

The `-events` option allows B to process inbound `dp_RPC` calls, but B will be unresponsive to other Tk events, such as events from the window system (e.g., requests to redraw the screen) and timer events (created using the `after` command in Tk).

To make B responsive to other events while in an RPC, the `-events` option can be passed a list of event types which B should continue to process while waiting for the RPC to return. Table 1 lists the event types that can be processed with `-events`. For example, to force B to continue processing events generated by the X window system while waiting for a response from A, the `x` event type is used:

```
B% dp_RPC $server -events x Subscribe
```

To process timer events, use `timer` as the event type:

```
B% dp_RPC $server -events timer Subscribe
```

To process multiple event types, such as RPCs and X events, pass a list to `-events`.

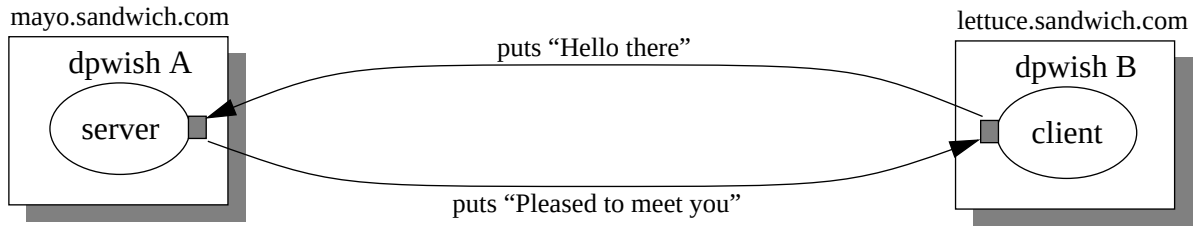

Figure 2: Deadlock

`Subscribe` procedure that clients may call to subscribe to the database which will build this list.

```
A% set clientList {}
A% proc Subscribe {} {
      global dp_rpcFile clientList
      if {[lsearch $clientList $dp_rpcFile] == -1} {
         lappend clientList $dp_rpcFile
      }
   }
```

After one or more clients have subscribed, the server can use the following code to broadcast a message to all its subscribers:

```
A% proc Publish {msg} {
      global clientList
      foreach c $clientList {
         dp_RPC $c puts $msg
      }
   }
```

This example brings up a subtle point that can cause your client/server application to deadlock. To illustrate, suppose A contains the following procedure:

```
A% proc Greeting {msg} {
      global dp_rpcFile
      puts $msg
      dp_RPC $dp_rpcFile puts “Pleased to meet you”
   }
```

Now suppose B issues the following call:

```
B% dp_RPC $server Greeting “Hello there”
```

The expected behavior is that “Hello there” will appear in A’s window, and “Pleased to meet you” will appear on B’s. Instead, both A and B hang because, while waiting for `dp_RPC` to return, B blocks. But, while processing the `Greeting` call, A issues a `dp_RPC` to B which cases it to block. Since B is stopped waiting for A, and A is stopped waiting for B, the system is *deadlocked*, as shown in figure 2.[2]

Tcl-DP provides three mechanisms to prevent deadlock

- the `-events` option of `dp_RPC`
- the `-timeout` option of `dp_RPC`, and
- the `dp_RDO` procedure.

---

2. If you run this example, your processes won’t actually deadlock because Tcl-DP uses a default value for the `-events` option of `dp_RPC`, which is discussed below, that prevents deadlock.

```
B% dp_RPC $server set x 5
5
B% set y [dp_RPC $server expr {8*$x}]
40
```

The curly braces in the `dp_RPC` call are needed to prevent the Tcl interpreter in B from substituting the local value of `x`, which contains the first line of `/etc/passwd`.

If an error occurs while executing an RPC, `dp_RPC` sets the `errorInfo` and `errorCode` variables in the originating interpreter and returns with an error. For example, the following call to `dp_RPC` triggers an error since `ReadFirstLine` requires a file name as a parameter.

```
B% set line1 [dp_RPC $server ReadFirstLine]
no value given for parameter “filename” to “ReadFirstLine”
```

The error is signaled using the standard Tcl mechanisms, exactly as if you had called `ReadFirstLine` locally. For example, the error can be trapped using the Tcl `catch` command:

```
if [catch {dp_RPC $server ReadFirstLine /does/not/exist} line1] {
   # Handle error any way you want...
   puts “Caught error: $errorInfo”
   puts “line1: $line1”
}
Caught error: couldn’t open “/does/not/exist”: No such file or directory
    while executing
“open $filename r”
    invoked from within
“set f [open $filename r]...”
    (procedure “ReadFirstLine” line 2)
    invoked from within
“ReadFirstLine /does/not/exist”
    invoked from within
“dp_RPC $server ReadFirstLine /does/not/exist”
line1: couldn’t open “/does/not/exist”: No such file or directory
```

The examples thus far have shown how a client uses `dp_RPC` to execute a command in a server. Now suppose the server needs to execute a command in a client. This function might be used, for instance, to build an application that supports a publish/subscribe paradigm. Clients contact the server to subscribe to a database, and servers issue callbacks to the clients when the database is updated. The server can use `dp_RPC` to issue such a callback, but in order to do so the server needs a connection identifier such as `$server`. Where does the server get the identifier for a client? The answer is from the `dp_rpcFile` variable.

Whenever Tcl-DP processes an RPC, it sets a global variable, named `dp_rpcFile`, to the connection identifier of the incoming RPC for the duration of the call. Servers can use `dp_rpcFile` to identify the source of the call, which can be used to contact the client later.

For example, suppose you want to write a server that supports the publish/subscribe paradigm. The server must maintain a list of all clients that have subscribed. The following code uses a

connection between the client and the server. The exact value of the identifier may be slightly different on your machine.

When A receives the connection request on port 4567, it opens a new file in the server that handles incoming `dp_RPC` requests. This leaves port 4567 free for accepting requests from other clients. Figure 1(B) show the machine and process architecture after the connection is established.

You can execute a Tcl command in the remote interpreter using this identifier as an argument to the `dp_RPC` command. For example, the following command prints "hello" in A's window:

```
B% dp_RPC $server puts hello
```

The extra arguments to `dp_RPC` (after `$server`) can be any Tcl command. For example, the following RPC creates a procedure in A that returns the first line in a file.

```
B% dp_RPC $server proc ReadFirstLine {filename} {
       set f [open $filename r]
       set firstline [gets $f]
       close $f
       return $firstline
   }
```

If B executes the following command, the variable `x` in B will contain the first line of the file `/etc/passwd` on A.

```
B% set x [dp_RPC $server ReadFirstLine /etc/passwd]
root:r.shdrfURbfwu:0:0:Operator:/:/bin/csh
```

This example shows an important feature of `dp_RPC`: the value returned by the `dp_RPC` call is the value returned by the command executed on A. As another example, the following sequence of commands creates a variable `x` in A, computes 8 times its value, and assigns the result to the variable `y` in B:

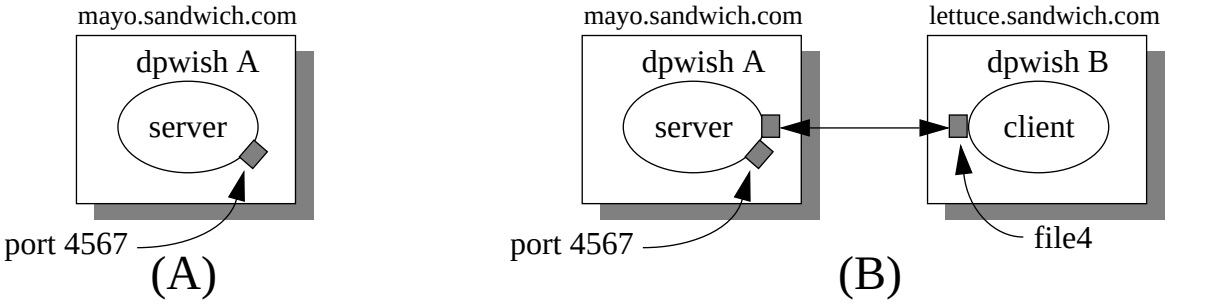

Figure 1: Connection setup

```
proc PromptB {} {puts -nonewline “B% “}
set tcl_prompt1 PromptB
```

The prompt for A and B should now be “A%” and “B%”, respectively. In the examples that follow, the prompt indicates in which `dpwish` the example commands are to be executed. We will refer to the processes as A and B.

The remainder of this chapter is divided into three sections. The first section summarizes Tcl-DP functions for creating client/server applications. After reading this section, you will be able to write robust distributed applications using Tcl-DP. The second section describes the socket level communication primitives in Tcl-DP. After reading this section, you will know how to use sockets and event handling. The third section describes the Tcl-DP distributed object system.

We assume that the reader is already familiar with Tcl/Tk.If not, the books by John Ousterhout and Brent Welch [ref,ref] provide excellent introductions. We also assume the user is familiar with the basic properties of Internet protocols like TCP/IP and UDP/IP, and has a superficial understanding of the Berkeley socket abstraction. The book by Stevens [ref] provides more information than you need to know on these topics.

Each section alternates between presenting a group of Tcl-DP functions and integrating them into an example program (a distributed whiteboard) that shows them in use.

Client/Server Architectures in Tcl-DP

The most important feature of Tcl-DP is that it simplifies the creation of client/server applications. For example, the following commands make A a server:

```
A% dp_MakeRPCServer 4567
4567
```

`Dp_MakeRPCServer` turns a process into a server listening on port 4567. The system will select a port number for you if you omit the port number or specify 0 as the port number. The chosen port number is returned, whether or not you specify it. The machine and process configuration is show in figure 1(A).

A client connects to a Tcl-DP server using the `dp_MakeRPCClient` command. `Dp_MakeRPCClient` takes two arguments: 1) the machine on which the server is running and 2) the port number on which the server will listen for client connection requests. For example, suppose the name of the machine that A is running on is `mayo.sandwich.com`. The following command will make B a client of A.

```
B% set server [dp_MakeRPCClient mayo.sandwich.com 4567]
file4
B%
```

The return value of `dp_MakeRPCClient`, `file4` in this example, is an identifier for the

Any command or script can be substituted in place of the `GetId` command. For example, the commands

```
dp_RPC $server info tclversion
dp_RPC $server info procs dp_*
```

return the version of Tcl that is running in the server process and all the Tcl-DP procedures in the server, respectively. Below, we will describe how a server can limit what machines can connect to it and what commands a client can execute.

`Dp_RPC` is similar to the `send` command in Tk. The primary difference is that `send` requires both processes to be connected to an X server to communicate, while `dp_RPC` can be run without an X server. Because `dp_RPC` does not use the X server for communication, it's faster than `send` -- 3 to 5 times faster for most commands.

Getting started

This chapter is designed to be used interactively. That is, although you can just read the chapter, you will get more out of it by trying out the commands as you read them. In order to run Tcl-DP scripts, you must run a `wish` that has been extended with Tcl-DP. This extension can be retrieved from

```
ftp://ftp.cs.cornell.edu/pub/tcl-dp/tcl-dp3.3.tar.gz
```

The distribution includes source code, instructions and scripts to configure, compile, and install the system, and Unix manual pages and several examples that document the system. The files README and INSTALL describe the distribution and how to make it.

Once installed, you can use the shell application called `dpwish` to try out the commands in this chapter. Type the command

```
dpwish
```

to your shell to invoke `dpwish`, which behaves like an ordinary `wish` interpreter, reading commands from standard input and writing the results to standard output.

Since Tcl-DP is intended for communicating applications, a second `dpwish` simplifies the examples. In another window on your machine, start up a second `dpwish`. We will call the first interpreter "A" and the second "B." To help you distinguish the interpreters, we recommend that you change the prompt of each interpreter. In A, use the following Tcl commands:

```
proc PromptA {} {puts -nonewline "A% "}
set tcl_prompt1 PromptA
```

Use these commands in B:

An Introduction To Tcl-DP

Brian Smith, Cornell University (bsmith@cs.cornell.edu)
Lawrence A. Rowe, University of California at Berkeley (larry@cs.berkeley.edu)

This document describes the Distributed Programming extension to Tcl/Tk, called Tcl-DP. Tcl-DP is a scripting language for writing client/server applications using Internet protocols and sockets. As with Tcl, the goal is ease of programming for applications, not maximal performance. In particular, Tcl-DP provides the following features:

1. Reliable Remote Procedure Call (RPC)
2. Automatic cleanup on file close and program exit
3. A name server for locating, starting, and authenticating servers.
4. Event handling functions
5. Support for TCP, UDP, and IP-multicast transport protocols
6. Socket configuration primitives
7. Interfaces to DNS lookup functions that map machine names to IP addresses.

The following script will give you a feel for the power of Tcl-DP. It uses Tcl-DP's RPC functions to implement a trivial "id server" that returns unique identifiers in response to `GetID` requests:

```
set myId 0
proc GetId {} {
   global myId
   incr myId
   return $myId
}
dp_MakeRPCServer 4545
```

All of the code in this script except the last line is ordinary Tcl code. It defines a global variable `myId` and a procedure `GetId` that increments the variable and returns the next id. The `dp_MakeRPCServer` command is part of Tcl-DP; it causes the application to listen for client requests on a TCP socket (port 4545)[1].

Other Tcl applications communicate with this server using scripts that look like the following:

```
set server [dp_MakeRPCClient server.company.com 4545]
dp_RPC $server GetId
```

The first command opens a connection to the id server and saves a reference to the connection in the variable `server`. The arguments to `dp_MakeRPCClient` identify the server's host and the port on which the server is listening. The `dp_RPC` command, whose arguments are a connection and an arbitrary Tcl command, performs a remote procedure call. `Dp_RPC` forwards this command to the server, which executes the script and returns a result (a new id in this case). The value returned by the server is the value returned by the `dp_RPC` command.

---

1. All commands in the Tcl-DP extension begin with "dp_"