CHAPTER 6

Writing Request/Response Clients and Servers

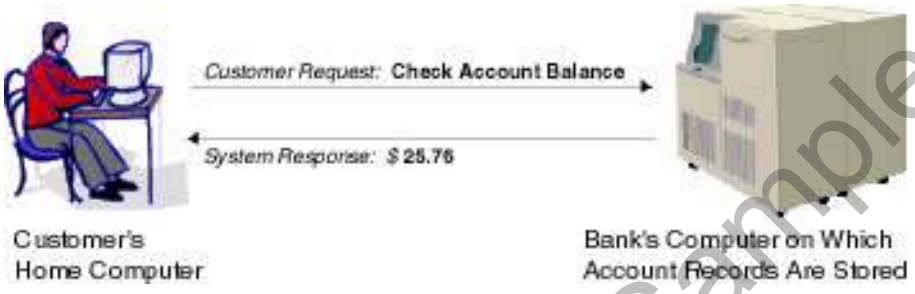
This topic includes the following sections:

Overview of Request/Response Communication

In request/response communication mode, one software module sends a request to a second software module and waits for a response. Because the first software module performs the role of the client, and the second, the role of the server, this mode is also referred to as client/server interaction. Many online banking tasks are programmed in request/response mode. For example, a request for an account balance is executed as follows in Figure 6-1:

1. A customer (the client) sends a request for an account balance to the Account Record Storage System (the server).
2. The Account Record Storage System (the server) sends a reply to the customer (the client), specifying the dollar amount in the designated account.

Figure 6-1 Example of Request/Response Communication in Online Banking



Once a client process has joined an application, allocated a buffer, and placed a request for input into that buffer, it can then send the request message to a service subroutine for processing and receive a reply message.

Sending Synchronous Messages

The `tpcall(3c)` function sends a request to a service subroutine and synchronously waits for a reply. Use the following signature to call the `tpcall()` function:

```
int
tpcall(char *svc, char *idata, long ilen, char **odata, long *olen, long
flags)
```

Table 6-1 describes the arguments to the `tpcall()` function.

Table 6-1 tpcall( ) Function Arguments

Argument	Description
svc	Pointer to the name of the service offered by your application.
idata	Pointer that contains the address of the data portion of the request. The pointer must reference a typed buffer that was allocated by a prior call to `tpalloc()`. Note that the `type` (and optionally the `subtype`) of *idata* must match the `type` (and optionally the `subtype`) expected by the service routine. If the types do not match, the system sets `tperrno` to `TPEITYPE` and the function call fails. If the request requires no data, set *idata* to the NULL pointer. This setting means that the parameter can be ignored. If no data is being sent with the request, you do not need to allocate a buffer for *idata*.

Table 6-1 tpcall( ) Function Arguments

Argument	Description
ilen	Length of the request data in the buffer referenced by *idata*. If the buffer is a self-defining type, that is, an FML, FML32, VIEW, VIEW32, X_COMMON, X_C_TYPE, or STRING buffer, you can set this argument to zero to indicate that the argument should be ignored.
**odata*	Address of a pointer to the output buffer that receives the reply. You must allocate the output buffer using the tpalloc() function. If the reply message contains no data, upon successful return from tpcall(), the system sets **olen* to zero, and the pointer and the contents of the output buffer remain unchanged. You can use the same buffer for both the request and reply messages. If you do, you must set **odata* to the address of the pointer returned when you allocate the input buffer. It is an error for this parameter to point to NULL.
olen	Pointer to the length of the reply data. It is an error for this parameter to point to NULL.
flags	Flag options. You can OR a series of flags together. If you set this value to zero, the communication is conducted in the default manner. For a list of valid flags and the defaults, refer to tpcall(3c) in the *Oracle Tuxedo ATMI C Function Reference*.

tpcall() waits for the expected reply.

Note: Calling the tpcall() function is logically the same as calling the tpacall() function immediately followed by tpgetrply(), as described in "Sending Asynchronous Messages" on page 6-11.

The request carries the priority set by the system for the specified service (*svc*) unless a different priority has been explicitly set by a call to the tpsprio() function (described in "Setting and Getting Message Priorities" on page 6-16).

tpcall() returns an integer. On failure, the value of this integer is -1 and the value of tperrno(5) is set to a value that reflects the type of error that occurred. For information on valid error codes, refer to tpcall(3c) in the *Oracle Tuxedo ATMI C Function Reference*.

Note: Communication calls may fail for a variety of reasons, many of which can be corrected at the application level. Possible causes of failure include: application defined errors (TPESVCFAIL), errors in processing return arguments (TPESVCERR), typed buffer errors

(TPEITYPE, TPEOTYPE), timeout errors (TPETIME), and protocol errors (TPEPROTO), among others. For a detailed discussion of errors, refer to "Managing Errors" on page 11-1. For a complete list of possible errors, refer to tpcall(3c) in the *Oracle Tuxedo ATMI C Function Reference*.

The Oracle Tuxedo system automatically adjusts a buffer used for receiving a message if the received message is too large for the allocated buffer. You should test for whether or not the reply buffers have been resized.

To access the new size of the buffer, use the address returned in the **olen* parameter. To determine whether a reply buffer has changed in size, compare the size of the reply buffer before the call to tpcall() with the value of **olen* after its return. If **olen* is larger than the original size, the buffer has grown. If not, the buffer size has not changed.

You should reference the output buffer by the value returned in *odata* after the call because the output buffer may change for reasons other than an increase in buffer size. You do not need to verify the size of request buffers because the request data is not adjusted once it has been allocated.

Note: If you use the same buffer for the request and reply message, and the pointer to the reply buffer has changed because the system adjusted the size of the buffer, then the input buffer pointer no longer references a valid address.

Example: Using the Same Buffer for Request and Reply Messages

Listing 6-1 shows how the client program, audit.c, makes a synchronous call using the same buffer for both the request and reply messages. In this case, using the same buffer is appropriate because the *audv message buffer has been set up to accommodate both request and reply information. The following actions are taken in this code:

1. The service queries the b_id field, but does not overwrite it.
2. The application initializes the bal and ermsg fields to zero and the NULL string, respectively, in preparation for the values to be returned by the service.
3. The svc_name and hdr_type variables represent the service name and the balance type requested, respectively. In this example, these variables represent account and teller, respectively.

Listing 6-1 Using the Same Buffer for Request and Reply Messages

```
. . .
/* Create buffer and set data pointer */

audv = (struct aud *)tpalloc("VIEW", "aud", sizeof(struct aud));

       /* Prepare aud structure */

audv->b_id = q_branchid;
audv->balance = 0.0;
(void)strcpy(audv->ermsg, "");

       /* Do tpcall */

if (tpcall(svc_name,(char *)audv,sizeof(struct aud),
       (char **)&audv,(long *)&audr1,0)== -1){
       (void)fprintf (stderr,"%s service failed\n %s: %s\n",
       svc_name, svc_name, audv->ermsg);
       retc = -1;
}
else
       (void)printf ("Branch %ld %s balance is $%.2f\n",
           audv->b_id, hdr_type, audv->balance);
. . .
```


Example: Testing for Change in Size of Reply Buffer

Listing 6-2 provides a generic example of how an application test for a change in buffer size after a call to `tpcall()`. In this example, the input and output buffers must remain equal in size.

Listing 6-2 Testing for Change in Size of the Reply Buffer

```
char *svc, *idata, *odata;
long ilen, olen, bef_len, aft_len;
. . .
```

```
if (idata = tpalloc("STRING", NULL, 0) == NULL)
   error

if (odata = tpalloc("STRING", NULL, 0) == NULL)
   error

place string value into idata buffer

ilen = olen = strlen(idata)+1;
. . .
bef_len = olen;
if (tpcall(svc, idata, ilen, &odata, &olen, flags) == -1)
   error

aft_len = olen;

if (aft_len > bef_len){  /* message buffer has grown */

   if (idata = tprealloc(idata, olen) == NULL)
      error
}
```

Example: Sending a Synchronous Message with TPSIGRSTRT Set

Listing 6-3 is based on the `TRANSFER` service, which is part of the `XFER` server process of `bankapp`. (`bankapp` is a sample ATMI application delivered with the Oracle Tuxedo system.) The `TRANSFER` service assumes the role of a client when it calls the `WITHDRAWAL` and `DEPOSIT` services. The application sets the communication flag to `TPSIGRSTRT` in these service calls to give the transaction a better chance of committing. The `TPSIGRSTRT` flag specifies the action to take if there is a signal interrupt. For more information on communication flags, refer to `tpcall(3c)` in the *Oracle Tuxedo ATMI C Function Reference*.

Listing 6-3 Sending a Synchronous Message with TPSIGRSTRT Set

```
    /* Do a tpcall to withdraw from first account */

if (tpcall("WITHDRAWAL", (char *)reqfb,0, (char **)&reqfb,
    (long *)&reqlen,TPSIGRSTRT) == -1) {
    (void)Fchg(transf, STATLIN, 0,
    "Cannot withdraw from debit account", (FLDLEN)0);
    tpfree((char *)reqfb);
}
...
    /* Do a tpcall to deposit to second account */

if (tpcall("DEPOSIT", (char *)reqfb, 0, (char **)&reqfb,
    (long *)&reqlen, TPSIGRSTRT) == -1) {
    (void)Fchg(transf, STATLIN, 0,
    "Cannot deposit into credit account", (FLDLEN)0);
    tpfree((char *)reqfb);
}
```


Example: Sending a Synchronous Message with TPNOTRAN Set

Listing 6-4 illustrates a communication call that suppresses transaction mode. The call is made to a service that is not affiliated with a resource manager; it would be an error to allow the service to participate in the transaction. The application prints an accounts receivable report, `accrcv`, generated from information obtained from a database named `accounts`.

The service routine `REPORT` interprets the specified parameters and sends the byte stream for the completed report as a reply. The client uses `tpcall()` to send the byte stream to a service called `PRINTER`, which, in turn, sends the byte stream to a printer that is conveniently close to the client. The reply is printed. Finally, the `PRINTER` service notifies the client that the hard copy is ready to be picked up.

Note: The example “Sending an Asynchronous Message with TPNOREPLY | TPNOTRAN” on page 6-13 shows a similar example using an asynchronous message call.