Programming a Distributed Application

The TUXEDO® System Approach

PREVIEW

This paper describes the rich set of communication techniques available to TUXEDO System programmers for writing distributed applications. The TUXEDO System supports four distinct communication methods that are versatile and easy to use yet powerful enough to build a wide variety of mission-critical business applications.

NOVELL

Programming a Distributed Application
The TUXEDO® System Approach

1. Introduction

A distributed computer application consists of a number of computers and software modules that run on them. What ties these computers together into a single application is the fact that the application software modules on them communicate with each other to perform the tasks required of the application. For example, a simple Automated Teller System may consist of the computers that are inside the automated teller machines (ATMs), plus the computer where the bank's records are stored. Figure 1 depicts this system. For the purposes of this paper, we won't distinguish between an ATM and its software component. We'll call both ATM, although most of the time we will be discussing the software. Likewise we use Record Storage System to denote both the computer and its software.

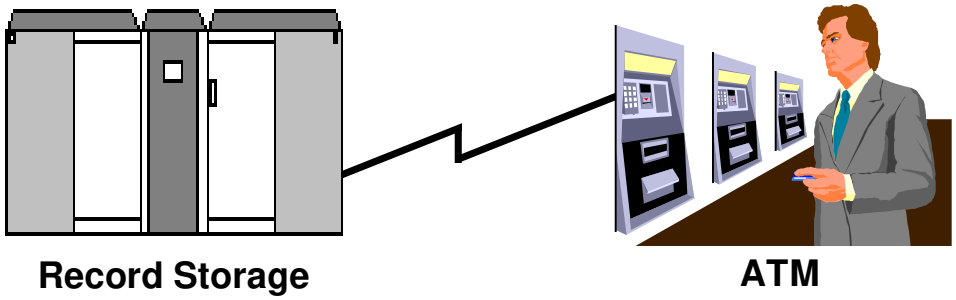

Figure 1. Automated Teller System Configuration

When a customer makes a request at an ATM a number of things might occur. The ATM might:

- verify the validity and currency of the customer's ATM card by communicating with the Record Storage System, or
- formulate a balance inquiry, deposit or withdrawal request and send it to the Record Storage System.

These actions include communication between the ATM and the Record Storage System. There are a number of other interactions that might take place between these two systems, including, for example, administrative communications when either of them goes off-line. Together, the modules in the ATM and in the Record Storage System, and the communication among them, comprise a distributed application, in this case an Automated Teller System.

We begin by describing the variety of ways in which the components of a distributed application communicate with each other. Next, we present the programming models supported by the TUXEDO System's Application Programming Interface (API) and show how this API can be used to write distributed applications.

1.1 Events - One Way Communication

The simplest communication between two modules is when one "tells" the other something. This is done by sending a message from one to the other. Presumably, the module receiving the message does something as a result of receiving it. For example, in the Automated Teller System when an ATM is started up, it might send a notification to the Record Storage System that it has come on-line (Figure 2). In effect, one module is conveying to another that some "event" has happened, and that the recipient of the "event" should "take due notice".

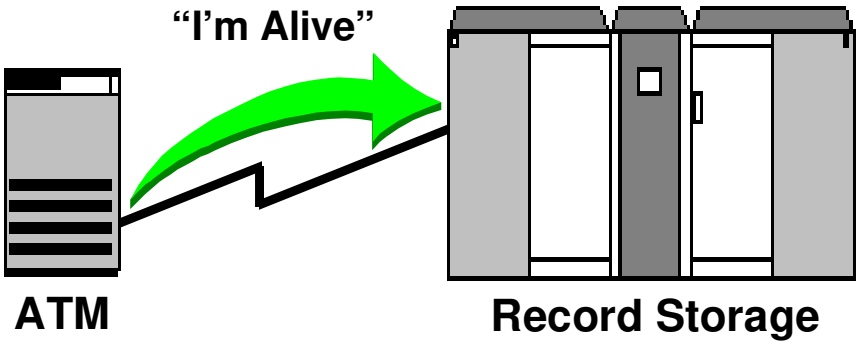

Figure 2. ATM sends an event message to Record Storage

1.2 Request/Response

If the module sending the message is expecting a reply from the module to which the message is being sent, then it is necessary for the receiver of the message to send back an answer. This kind of interaction is so common that it has been given a name: Request/Response. The sender makes a request in the form of a message it sends, and the receiver processes it and sends back a response message. We also term this type of interaction *client/server*, because the module receiving the message services requests sent to it by client modules, which are asking that the service be done.

For example, in the Automated Teller System, when the user presents an ATM card and types in the Personal Identification Number (PIN), the ATM asks the Record Storage System if the card is valid (i.e., that the person presenting the card has entered the correct PIN, and that the person is still a customer in good standing). That is, the ATM makes a validation request of the Record Storage System. The Record Storage System responds with "yes" or "no" (Figure 3). Depending on the response from the Record Storage System to its validation request, the ATM either accepts more input from the customer or rejects the customer. After validation, the customer might ask for an account balance. This becomes another request/response interaction between the ATM and the Record Storage System.

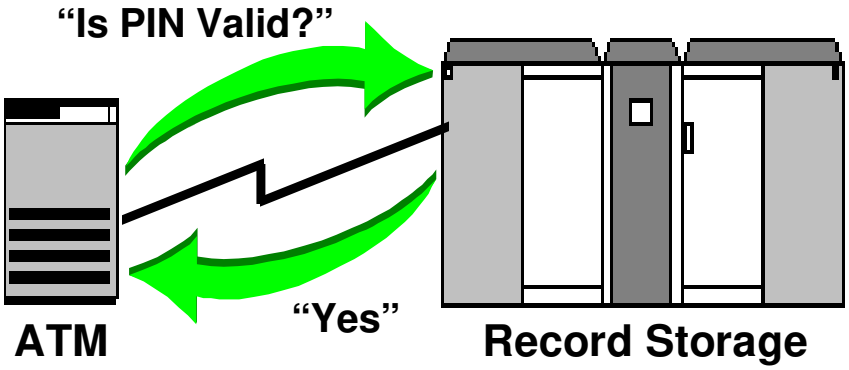

Figure 3. ATM and Record Storage having a request/response exchange

1.3 Conversational Interactions

Request/Response is a simple type of dialogue. The rules for communication during request response, that is, the "protocol" for exchanging information, are fixed: the client asks something, the server responds. The client never sends more than one message as part of its request, nor does the server send multiple responses in reply. A generalization of request/response is a dialogue, or conversation, between software modules. Here, the modules have an extended amount of information to exchange, and their "communication protocol" may be more complicated.

For example, a customer in the Automated Teller System might ask for a list of all account activity made that month (Figure 4). The answer may be information on numerous accounts and deposits, and the server may have many replies to send back. As the ATM might have only limited memory capacity, the Record Storage System may send back only some of the information, and await the ATM's request for "more". This protocol could have multiple messages replying to each "more" request. Additionally, the server needs to keep track of what it has already sent back, so a request from the ATM for "more" data can be handled by giving it another set of account-activities that the ATM hasn't yet "seen".

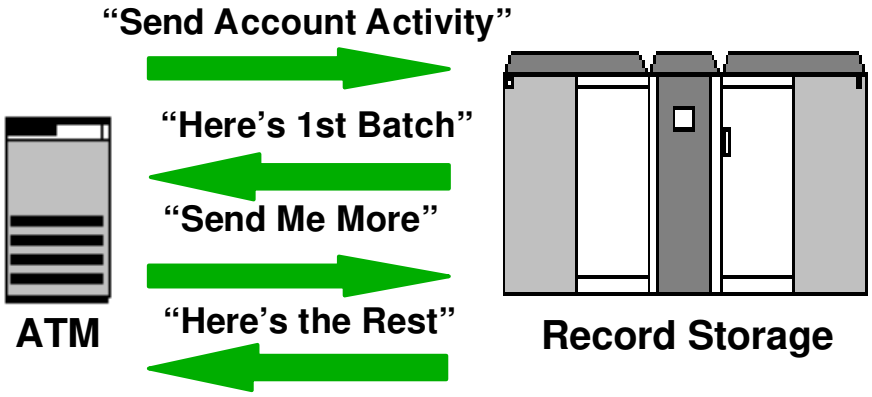

Figure 4. ATM and Record Storage having a conversation

1.4 Queued Communications

It is not always the case that a partner module is available for communication. Perhaps the Record Storage System in the Automated Teller System is "down". Can the Automated Teller System still run? Depending on the rules of the bank, some functions may be available even with the Record Storage System down. Assuming the ATM can "quick validate" PINs without the Record Storage System (e.g., just validate that the correct password, whose encoded form is on the card, was entered correctly, but not validate that the customer is still in good standing), the ATM can accept deposits. The ATM may, in fact, be allowed to honor withdrawals (e.g., for small amounts of money) even when the Record Storage System is down.

A key requirement, however, is that when the Record Storage System becomes available, the ATM can send the Record Storage System information on actions (i.e., deposits and withdrawals) it has performed while the Record Storage System was unavailable. In order to do this, the ATM must record its actions for later transmission to the Record Storage System (Figure 5). This recording of actions must be done to permanent media (e.g., a disk), in case the ATM also goes down before the Record Storage System comes up. When the Record Storage System does come up, the ATM can send it all the requests it has processed and saved on disk. In effect, the ATM and the Record Storage System have communicated "off-line" via a disk-based message queue.

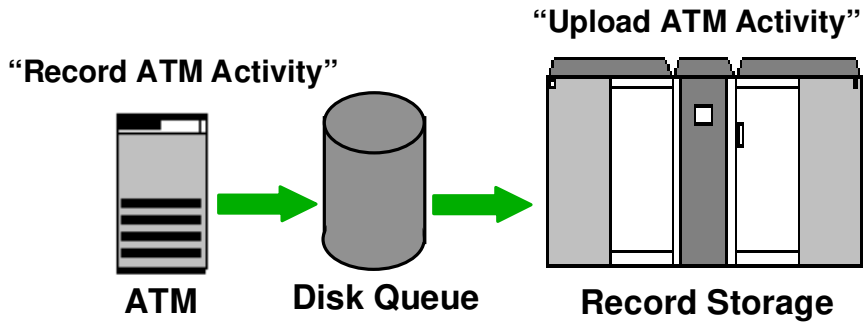

Figure 5. ATM and Record Storage communicating through disk-resident queue

1.5 Server/Client Communication - Events Revisited

In the Automated Teller System, what if the bank wants to take the Record Storage System off-line for a while (perhaps to reconcile/audit accounts)? How is the ATM to find out that the Record Storage System isn't available? One (ungraceful) way is to do nothing. When the ATM makes a request there will be no response, the ATM can "time-out" and tell the customer that something is amiss. There are number of problems with this: how long does the ATM wait, and how does it know if the Record Storage System's absence was planned or not?

Perhaps a better way is to have the Record Storage System tell the ATM that it's going off-line for a while. The ATM then "knows" what's happening and can give a more informed message to its customers. There's no waiting, no guessing, and if the Record Storage System gives the ATM an indication of how long it will be down, the ATM may even be able to tell the customer an estimate of when full service will be restored. This requires the server system to notify the client system, which might be in the process of dealing with a customer. Moreover, the server system isn't making a request of the client, it is merely notifying it of some event or circumstance that is taking place in the server. What is needed is a way for servers to notify clients of events happening at the server system, and for the client to accept such notification. This is what we previously referred to as "event" communication.

2. Requirements of an API for Distributed Systems

What should one look for in an Application Programming Interface (API) used for building distributed systems?

2.1 Ease of Use

The person writing a distributed application should be able to concentrate on the application. In particular the API should be simple to use, and powerful in effect. Simple to use means that low level details of the application's distribution should not be the concern of the programmer. Also, physical details of the distribution should be hidden: the programmer should be able to use logical names in the API, such as "Branch Office", rather than having to know the physical network addresses of the machines in the application. Doing so also makes it easier to physically reconfigure the hardware without having to change the software. Since distributed applications are susceptible to network failures, the application should be given a simple way to deal with these. For example, a time-out capability might be provided, so that remote calls don't "hang", and so that the application doesn't itself have to set timers and provide "watch dog" routines.

2.2 Suitability to Tasks

In the course of designing and implementing a distributed application, the API should allow one to use the various methods we've already covered: request/response, conversations, queued communication and events.

2.3 Error Handling

The most difficult part of programming is handling error cases. This is made worse in distributed systems, where it is not easy for applications to "understand" errors occurring in remote parts of the application that may be affecting their operation. For example, when a request is made and no response is forthcoming, what is the requester supposed to do? Did the request even get to its destination? If it did, has it been processed? If so, is it the response that is hung up? If the response was sent but has "been lost", the server module may "think everything is OK". Since the requester hasn't received a response, should it re-send the request? How many times? What if the request was to transfer $1,000,000 from one bank account to another? In this case, the programmer wants to be sure to "do the right thing"! A number of techniques may help the programmer:

- making the low level networking calls easy to use by providing higher level facilities (e.g., transparent application data conversion, transparent flow control, etc.),
- automatically timing out calls that are taking too long,
- making failure cases easy to deal with.

2.4 Effect of Transactions

A transaction is a way to make sure that a distributed application "does the right thing". That is because transactions ensure that the distributed application:

- does each (remote) operation exactly once, and
- does all or none (but not some) of a set of related calls.

If something "goes wrong" during a transaction, all of the effects that have been done during its lifetime are undone. The application programmer doesn't have to worry about partially completed actions. Undoing the effects of a transaction is called "rollback". The effect of a rollback is to "wipe the slate clean", thus allowing the programmer to abandon what had been attempted, or to retry it as part of a new transaction. Without transaction support being an integral part of a distributed programming API, programmers themselves must write additional code to perform any necessary "undoing" of partially completed actions (often referred to as "compensating actions").

2.5 Heterogeneous Systems

Today, many distributed systems exist in an environment of mixed hardware and software components. Moreover, the mix is changing all the time as new hardware innovations are made. While it is highly desirable that the deployers of an application have the freedom to choose the computing platforms on which to run it, it is also desirable to shield the application programmer from the peculiarities of particular hardware platforms, and also from the intricacies of making different platforms interoperate. In particular, the application programmer should not have to be concerned with:

- CPU architecture (i.e., how data items are represented)
- interconnection media (i.e., LANs, LAN protocols, WANs, etc.)

3. The TUXEDO System API

The TUXEDO System provides an API that not only offers a wide variety of communication methods, but it also makes writing distributed applications simpler by supporting, among other features, transactional communication and the hiding of hardware and networking details from programmers.

The TUXEDO System API offers both library-based and language-based programming. Library-based programming requires programmers to use a set of C or COBOL procedures defined by the TUXEDO System. The TUXEDO System's procedure library is known as the Application to Transaction Manager Interface, or simply, ATMI[1]. ATMI is a superset of X/Open's XATMI interface[2]. Procedures in the ATMI library start with the prefix "tp" (e.g., tpcall()). ATMI procedures can be used, for example, to send a request from one program to another, or to have one program enqueue data to stable storage for later retrieval by another.

The TUXEDO System's language-based programming paradigm is a remote procedure call (RPC) facility called TxRPC, the TUXEDO System's implementation of X/Open's TxRPC interface[3]. TxRPC is OSF's DCE RPC facility extended to support transactional communication. With RPCs, an program invokes application-defined procedures (e.g., atm_inquiry()) that execute in different programs, possibly on different computers.

Remote procedures are defined using an interface definition language (IDL) that contains the definition of the procedure's interface (i.e., its language bindings, its parameters and return value) as well as additional information required for distribution (e.g., version numbers, communication addresses, etc.). Thus, RPC is a form of distributed communication where the syntax is nearly identical to a procedure call and the called procedure is not co-located with the caller. Because calling a procedure is a local form of request/response (i.e., the caller invokes a procedure and awaits its completion along with any results), RPCs are an ideal mechanism for distributed request/response programming.

The table below gives an overview of the TUXEDO System's API components and the rest of this section discusses each method and its variations in more detail.

Request/Response	Conversational	Queuing	Events
Library-based: tpcall(), tpacall(), tpgetrply(), tpreturn(), tpforward() Language-based: TxRPC (OSF DCE IDL)	tpconnect(), tpsend(), tprecv(), tpdiscon()	tpenqueue(), tpdequeue()	Unsolicited: tpnotify(), tpbroadcast() Brokered: tpsubscribe(), tpunsubscribe(), tppost()

Table 1. Programming Models Supported in the TUXEDO System API

3.1 The Data

The content of data sent via ATMI includes simple strings, aggregate data types (i.e., C structures and COBOL records), an attribute/value data type called a fielded buffer, and user defined types. The data itself is sent in a "typed buffer", which is a data structure that identifies the data type being sent (e.g., C structure, string, fielded buffer), and contains the data itself.

The fielded buffer data type, which is accessed by a set of functions called FML, provides a facility for programmers to send a variable number of typed data items to a partner program. The actual structure of the data is hidden from the communicating programs, and access to the data is via function call. This facility is extremely powerful because:

- It allows programs to send variable amounts of data.

- It hides the data's format, and thus allows the addition of new data items. Programs that "don't know" about new data items remain insensitive to them.

User defined types allow programmers who must invent new data types to support complex applications to do so. These types may be incorporated into ATMI calls.

The following is an example of allocating a string typed buffer and populating it with a customer name:

```
char *str;

str = tpalloc("STRING", NULL, 120);  /* get a string typed buffer for input (120 bytes) */
 (void)  strcpy(str, cust_name);          /* put the name in the typed buffer */
```

Once populated, a typed buffer can be sent to other programs or to disk-based queues using the ATMI communications routines. The advantage of typed buffers is that programmers do not have to worry about converting data being sent to machines having different data representation formats. Using typed buffers, application data can be sent "as is". The TUXEDO System transparently performs data format conversions when necessary. Of course, both the sending and receiving programs must agree on the buffer types they will be exchanging.

Within TxRPC, data items are C-language data types as expressed in the OSF DCE Interface Definition Language (IDL). This syntax has been extended within the TUXEDO System's implementation to allow the specification of FML data buffers as parameters to functions. The IDL for the "deposit" and "withdraw" functions within the Automated Teller System might look like this:

```
[uuid(C996A680-9FC2-110F-9AEF-930269370000), version(1.0), transaction_optional]
interface ATS
{
int deposit([in] int account, [in] int amount, [out] int *balance);
int withdraw([in] int account, [in] int amount, [out] int *balance);
}
```

This IDL is compiled by an IDL compiler that emits "stubs" for use by both the client and the server. These stubs maintain the illusion of simple, local procedure calls but they contain the software necessary to transfer the parameters between client and server (known as "marshaling") and to perform the remote execution of the called procedures.

3.2 Request/Response

There are variations of request/response communication. The TUXEDO System's API supports both synchronous as well as asynchronous request/response. ATMI's tpcall() and TxRPC's Remote Procedure Call facility provide for synchronous processing while ATMI's tpacall() and tpforward() allow for parallel execution through asynchrony.

3.2.1 Synchronous Request/Response - TxRPC

As previously described, RPC is a very simple form of request/response because it models a local procedure call. With local procedures, the caller must block and wait for the called procedure to complete before regaining control of the CPU. Similarly, since RPCs mimic local procedure calls in syntax and semantics, RPCs are intrinsically synchronous and, therefore, have no asynchronous counterpart. The following shows the use of TxRPC within the Automated Teller System.

When the ATM performs a funds transfer from one account to another, the ATM runs a client program that calls RPCs that are offered on the Record Storage System. Assuming the IDL defined in the previous section, the ATM's code, which contains the client stubs, would look something like this:[1]

```
tx_begin();
    ret1 = withdraw(account1, amount1, &balance1);
    ret2 = deposit(account2, amount2, &balance2);
if (ret1 == SUCCESS && ret2 == SUCCESS)
    tx_commit();
else
    tx_rollback();
```

Notice that because both parts of the transfer must succeed or fail as a unit, the RPCs are performed within a transaction boundary. Thus, if either fail, both will rollback and the ATM will inform the customer that his/her transfer has failed.

The code that implements deposit() and withdraw() resides on the Record Storage System, along with the compiled IDL server stubs for invoking these functions, and would have this general appearance:

```
int deposit(int account, int amount, int *balance)
{
    /* perform SQL calls for deposit service */
    return(SUCCESS);
}

int withdraw(int account, int amount, int *balance)
{
    /* perform SQL calls for withdraw service */
    return(SUCCESS);
}
```

As is evident from our RPC example, all the communication software necessary to allow the ATM to communicate with the Record Storage System is hidden from the programmer.

3.2.2 Synchronous Request/Response - ATMI's tpcall()

In ATMI, request/response communication is accomplished via the function tpcall(). Similar to an RPC, tpcall() invokes an application-provided service routine, which may either be on the same computer or a different computer than its caller. tpcall() sends input data to the service routine, and gets output data from it. Unlike an RPC, tpcall() can also take options to direct the TUXEDO System on how to handle the request (e.g., "don't send the request if a blocking condition exists") and tpcall() can return both application- and system-defined return values (e.g., "the service routine completed such that the current transaction will be rolled back").

When invoking tpcall(), the client passes it the name of the service to be invoked, and two buffers. One buffer is input to the service, the other will contain the service's output (the input and output buffers may in fact be the same buffer, in which case the response from the service will append to or overwrite the request data sent to it). In writing a tpcall(), therefore, the programmer should be aware of the name of the service to be called, what data the service expects to receive to perform the service, and what the resulting output of the service will be. In effect, the requester and the service provider have a contract: if the requester supplies certain items to the service, the service will process them and respond with the results of its processing. The requester doesn't know where the server resides.

[1] The programming examples in this paper are illustrative and are not meant to be complete programs. While the TUXEDO ATMI and TxRPC syntax is intended to be correct, other logic required for complete and correct programs is not shown (e.g., data declarations and complete error handling).

The effect of a call to tpcall() is that the named service is invoked and passed the input data. It is the responsibility of the service to take the input data, perform the implied processing, and send back a response. The input is made available to the service in a buffer. The reply is also in the form of a buffer. The server sends back a reply to the requester by a call to the function tpreturn(). The service neither knows where the request came from, nor where the reply is going. tpreturn() has the following effects:

- it sends back the reply message to the requester,
- it provides the requester with an indication of the "success" or "failure", from the server's perspective, of the service processing,
- it concludes the execution of the service

The following example illustrates the client-side (ATM) of a tpcall() to validate a person presenting his/her card to an ATM. Assuming the typed buffer for the account information has been allocated, the ATM's call to the Record Storage System's validation service would look like:

```
/* call VALIDATE using account information buffer for both input and output */
acct_info ->acct = acct_number;
acct_info ->PIN = encrypt(PIN);
if (tpcall("VALIDATE'', acct_info, len, &acct_info, &ret_sz, TPNOBLOCK) == -1) {
        /* handle error (eg, communication failure,  PIN invalid, time-out, etc) */
}
/* PIN valid: service returns more fields in info buffer (eg, recently overdrawn, etc) */
```

Here is what the application service routine on the Record Storage System might look like:

```
VALIDATE(TPSVCINFO *request)
{
        /* call authorization routine and pass it the input data (ie, name and PIN) */
        data = request->data;
        if (authorize(data) == SUCCESS) {
            /* if success, return from service and pass back account info */
            get_cust_info(data);
            tpreturn(TPSUCCESS, 1, data, len, TPNOFLAGS);
        } else {
            /* note that even on application failure, can still pass back output data */
            add_failure_report_to_buffer(data);
            tpreturn(TPFAIL, -1, data, len, TPNOFLAGS);
         }
}
```

3.2.3 Asynchronous Request/Response (Fan-out Parallelism)

When tpcall() returns to its invoker, the service has been executed and the results of its processing are available. That is, the requester has been suspended while the service has been executed. In this case, we say that the requester and server are synchronized, or that the processing is synchronous. This is satisfactory if the requester has nothing else to do while the service is being executed, or if the requester needs the results of the service in order to proceed. Sometimes the client can usefully do another task while the server is processing its request. In fact, the requester may be able to invoke a different service while its first service request is still executing. Recall the funds transfer example above in the TxRPC scenario. In that example, the withdraw() function was called synchronously and had to complete before the deposit() function could be called.

These sub-functions may be independent, and in fact could be executed simultaneously (i.e., in parallel). If they are executed in parallel, the application calling them may see better response time than if they are executed one after the other (called "serial execution"). In order to accommodate this type of parallel

processing, ATMI provides a function called tpacall(). tpacall() acts just like tpcall(), with the exception that it does not wait for the service to be executed before returning to the caller. Instead, it returns while the service is executing, thereby allowing the caller to do additional work, including making another tpacall().

If the caller does perform another tpacall(), there will be two services executing in parallel. In this case we say that the client and server are acting asynchronously. Ultimately, the requester will want to get replies from the invoked services. This is done with a call to the function tpgetrply(). We call the type of parallelism afforded by tpacall() "Fan-out Parallelism", because actions "fan-out" from the invoking software module. The following shows a rewrite of the client-side our funds transfer example using tpacall() and tpgetrply():

```
tx_begin();
    handle1 = tpacall("WITHDRAW", acct_info1, len, TPNOFLAGS);
    handle2 = tpacall("DEPOSIT", acct_info2, len, TPNOFLAGS);
     /* The TPGETANY flag tells ATMI to get whichever reply arrives first. */
     /* On output, the program can compare reply1 and reply2 with */
      /* the above handles to match replies with requests */
     tpgetrply(&reply1, &output_buf1, &buf_sz1, TPGETANY);
     tpgetrply(&reply2, &output_buf2, &buf_sz2, TPGETANY);
tx_commit();
```

3.2.4 Asynchronous Request/Response (Pipeline Parallelism)

To improve throughput, modern CPUs are implemented with a technique called pipelining. In order to perform an instruction, a number of functions, called stages, must be performed by the CPU. The stages are separated into a set of sequential operations, with each operation being performed by separate electronic circuitry. Each stage's output becomes input to the next stage. In order not to have circuitry "being idle", stage execution is overlapped within the CPU. That is, when the first stage has completed its operation and passed its results on to its subsequent stage, the first stage begins working on the next instruction. Both stages are working at the same time.

ATMI enables "software pipelining". This feature may be used to improve throughput of the software modules in the system, by overlapping the work they do. The way this works is as follows. Instead of calling tpreturn() to respond to a request, a service, let's call it service "A", may call the function tpforward() to send the request on to another service, which we'll call service "B", for additional processing. After calling tpforward(), service "A" becomes free to begin work on a request from a different client. When service "B" calls tpreturn() the response is sent to the requester. In effect, "A" and "B" are two stages in the completion of the task requested by the client. The tpforward() function enables "pipeline parallelism" within a distributed application. It is the method by which results are passed between stages.

Here's how we might exploit tpforward() in the Automated Teller System. When a customer presents a card at the ATM, the ATM needs to perform two functions:

- validate the user's PIN, and
- ensure the customer has a "current account"

These functions are somewhat independent, and we might consider implementing them with some type of parallelism. However, before even attempting to check the currency of an account, we might want to validate the PIN. So, we could consider having the ATM call the Record Storage System's "VALIDATE" service. "VALIDATE" itself is really the first service of a two-stage pipeline. The first stage checks the password and passes it on to the second stage, called "CUST_STATUS". When "VALIDATE" gets a request from the ATM is does the PIN processing. If it fails, it responds (via tpreturn()) with a failure

response. If the PIN processing succeeds, "VALIDATE" forwards the work (via tpforward()) on to "CUST_STATUS". "CUST_STATUS" begins to check out the customer's status. Meanwhile, "VALIDATE" may be progressing on another request made from a different ATM. When "CUST_STATUS" completes, it responds to the ATM by calling tpreturn(). The following example shows how the client and the two services are structured. The client makes the same tpcall() as before (in fact, the client has no idea that pipelining is being performed by the servers):

```
tpcall("VALIDATE'', acct_info, len, &acct_info, &ret_sz, TPNOBLOCK);
```

The "VALIDATE" service is modified as follows:

```
VALIDATE(TPSVCINFO *request)
{
        /* call authorization routine and pass it the input data (ie, name and PIN) */
        data = request->data;
        if (authorize(data) == SUCCESS) {
            /* if success, forward request(and input data) to get the account status */
            tpforward("CUST_STATUS", data, len, TPNOFLAGS);
        } else {
            /* return immediately to client with failure report */
            add_failure_report_to_buffer(data);
            tpreturn(TPFAIL, -1, data, len, TPNOFLAGS);
         }
}
```

The "CUST_STATUS" service will complete the request and send the customer information record back to the client as its reply:

```
CUST_STATUS(TPSVCINFO *request)
{
        /* call status routine and pass it the input data (ie, authorized account) */
        data = request->data;
        if (get_cust_info(data) == SUCCESS) {
            /* if success, return account status info to client */
            tpreturn(TPSUCCESS, 1, data, len, TPNOFLAGS);
        } else {
            add_failure_report_to_buffer(data);
            tpreturn(TPFAIL, -1, data, len, TPNOFLAGS);
         }
}
```

Note that "CUST_STATUS" does not always have to be used as the second-half of "VALIDATE". It can be used as a standalone service. Thus, a collection of "elemental" services can be grouped together in a software pipeline to provide higher-level services, and better throughput, transparently to client applications.

3.3 Conversational Programming

For interactions between modules that require an extended interchange, ATMI provides a set of "conversational" functions.

- tpconnect() - allows a client to establish a connection to a service. The purpose of the connection is to send messages back and forth, in an analogous manner to a phone conversation. When the client calls tpconnect(), a server is notified (like the phone ringing!) via the invocation of a service routine and the connection is established.

- tpsend() and tprecv() are the functions used to send messages back and forth across the connection. As with tpcall(), the data content of the message is a typed buffer, which is converted by ATMI should the corresponding modules be on different machine types. In an ATMI conversation, only one program can send at a time. When that program is finished sending, it gives "send control" to the other program and then enters "receive mode". This is commonly referred to as a "half-duplex" conversation.
- Unlike tpcall() or RPC, "functions" aren't called on the server side. Instead, the service routine to which one is connected asks for a message (by calling tprecv()). This lets the service procedure easily keep context information in its local variables. As the conversation progresses the location of the sends and receives within the service routine also imply a context of the dialogue.
- Conversations are terminated when the service calls tpreturn() (due to the connection-oriented nature of conversations, tpforward() cannot be used by conversational services). However, if the client needs to terminate a conversation abruptly, it can use tpdiscon() to tear down the connection. In this case, the server will be informed of the disconnect, and any transactional context will be rolled back since the conversation didn't terminate "gracefully".

In the Automated Teller System, let's say that the ATM needed to download from the Record Storage System the valid list of branch account numbers once a week. It could connect to a conversational service on the Record Storage System that passed back to the client 100 accounts at a time. The ATM would process each set of 100 before asking for the next 100. The conversation might look like the following:

Client Code

```
records->batch_count = 100;       /* number of records service should return at a time */
/* connect to service procedure, send it the initial data, and give it send control */
cd = tpconnect("Get_Records", records, len, TPRECVONLY);
while (1) {    /* do forever */
        tprecv(cd, &records, &buf_sz, TPNOFLAGS, &revent);
        num_records_received = process_records(records);
        if (num_records_received < 100)
                break;
        /* flow control: informs service to send next set of records; no data sent */
        tpsend(cd, NULL, 0, TPRECVONLY, &revent);
}
```

Service Code

```
Get_Records(TPSVCINFO *msg)
{
        data = msg->data;
        while (records_left_to_send > data->batch_count) {
                get_next_batch_of_records(data);
                tpsend(msg->cd, data, len, TPRECVONLY, &revent);  /* give control to client */
                /* wait for client to send flow control asking for more records */
                tprecv(msg->cd, &data, &len, TPNOFLAGS, &revent);
        }
        /* send last batch as part of tpreturn() */
        get_last_batch_of_records(data);
        tpreturn(TPSUCCESS, 1, data, len, TPNOFLAGS);   /* gracefully terminates connection */
}
```

3.4 Queuing

Sometimes it is impossible for clients to communicate in an on-line manner with servers because the servers are not available, or are too busy to give a timely response. In such cases, it may be useful for clients to store their requests and have them processed when the server becomes available. The TUXEDO System supports such batch storage and retrieval of messages through its stored messaging facility, called /Q. Messages are stored in disk-resident message queues (and hence the name /Q) via a call to the function tpenqueue(), and may later be retrieved by a call to the function tpdequeue(). Responses from the server to the requester (which may have since disappeared) may also be routed through a response queue. The format of data that may pass through /Q is the same as for tpcall().

The /Q mechanism also provides a "forwarding agent" that converts enqueued messages to service requests. In fact, a server receiving such a request can't tell the difference between requests sent to it by tpcall() or requests forwarded to it by /Q's forwarder. /Q's forwarder dequeues requests using tpdequeue(), it then invokes tpcall() to forward the requests to servers. The forwarder does so under transaction control so that if a service routine is unavailable or fails, the action of tpdequeue() is rolled back causing the request to remain on disk. The /Q forwarder is configurable with respect to the number of retries it will attempt.

Within the Automated Teller System, if the Record Storage System goes off-line, the ATM could call tpenqueue() to save records of deposits/withdrawals at the local branch. When the Record Storage System comes on-line, the ATM can dequeue the messages and send them to the Record Storage System. Alternatively, /Q's forwarder can automatically dequeue the stored requests and forward them to servers on the Record Storage System.

3.5 Event Communication

In most applications, some event occurs that changes the state of the business. Mostly, these events are common to the functioning of the business and are planned for. In fact, request/response communication is setup to handle predetermined "events", called requests, and servers exist solely to "respond" to them. However, there are times when events occur where, instead of sending requests, unsolicited messages are sent to one or more programs to inform them of the events. For example, if the Record Storage System is coming down for maintenance, it might want to send this information to the ATM. The ATM might respond to this event by enqueuing all future requests to its local disk until it receives another message telling it that the Record Storage System is back on-line. Notice that these messages are not requests; rather, they are a form of event communication.

ATMI offers two types of event communication: unsolicited and brokered.

3.5.1 Unsolicited Events

In the unsolicited approach, client programs register an application routine with the ATMI library that they would like to have invoked when unsolicited messages are sent to them. ATMI supports two functions at the server to send unsolicited messages to clients. tpbroadcast() sends a message to one or more clients by the clients' "name". tpnotify() sends a message to a single client by its unique "client id". Note that since the client wasn't expecting anything from the server, it was likely doing something else, like communicating with an external device (as the ATM does in the Automated Teller System). So, it needs to be notified that the server is communicating with it. This ATMI does either by raising a signal in the client, or, more gently, by informing the client the next time the client makes an ATMI call.

Of course, it is possible to combine this style of server/client event communication and client/server communication within an application. For example, the server might tpbroadcast() an event to a client, which may then use tpcall() to get more information about the event.

Within the Automated Teller System, we might use tpnotify() to have "VALIDATE" alert a supervisory console if someone fails PIN processing more than three times in a row. For example, the program in control of the supervisory console would set-up to receive unsolicited notifications in the following way:

```
/* tell ATMI the name of the function to invoke when an unsolicited message arrives */
tpsetunsol(atm_handler);
```

For this to work, the client program should have defined a routine called atm_handler() that takes a typed buffer as its argument. Next is our modified "VALIDATE" service that notifies the supervisory console when a bad PIN is given more than three times:

```
VALIDATE(TPSVCINFO *request)
{
        /* call authorization routine and pass it the input data (ie, name and PIN) */
        data = request ->data;
        if (authorize(data) == SUCCESS) {
            /* if success, forward request(and input data) to get the account status */
            tpforward("CUST_STATUS", data, len, TPNOFLAGS);
        } else {
            if (auth_failed_count(data) >= 3)
                tpnotify(supervisor_console, data, len, TPNOFLAGS);
            add_failure_report_to_buffer(data);
            tpreturn(TPFAIL, -1, data, len, TPNOFLAGS);
         }
}
```

3.5.2 Brokered Events

In the brokered approach, application programs subscribe (with an event broker) to receive notification when a particular application event occurs. Other programs are allowed to post that an event has occurred. The event broker will inform all subscribers of that event. Subscribers use ATMI's tpsubscribe() to register with the TUXEDO event broker the events they are interested in "hearing" about. tpsubscribe() allows the caller to specify how notification should occur. For example, a server may want to receive a service request when event X occurs, but it may want to find out about event Y by using tpdequeue() to retrieve the event's existence from a disk-based queue. Programs can unsubscribe to events using tpunsubscribe(). Lastly, programs can inform the broker that an event has occurred using tppost().

An important difference between brokered and unsolicited event communication is that, in the former, event subscribers have no knowledge of event posters and vice versa. All communication is through an intermediary broker (similar to the queuing model whose intermediary is a queue). Thus, it is perfectly all right for posters to post events for which there are no subscribers. One benefit of this anonymity is that subscriber lists can grow or shrink completely unknown to the poster (i.e., the poster's code does not have to change when a new set of users join the application and begin subscribing to events).

Within the Automated Teller System, we could modify our "VALIDATE" service even further to call tppost() upon successive PIN failures. In this case, other programs, in addition to the supervisor, could also be notified without the server having to know who they are.

4. Conclusion

In attempting to implement applications for distributed business operations, a powerful, yet simple to program interface is needed by application programmers. The TUXEDO System's API provides a comprehensive yet comprehensible API for developers of distributed business applications. It supports four distinct communication paradigms: request/response, conversational, queuing and event communication. These are provided in the TUXEDO System in both library-based and language-based forms. Programmers using the TUXEDO API can pass data "as is" across heterogeneous environments without having to worry about converting data types for different machine representations. Also, the API

supports transaction semantics making error handling in distributed environments simple and manageable.

References

[1] Novell, “Transaction Monitor Reference Manual”. TUXEDO ETP System Release 4.2, Issue 4

[2] X/Open, “Distributed Transaction Processing: The XATMI Specification”, X/Open Preliminary Specification, July 1993

[3] X/Open, “Distributed Transaction Processing: The TxRPC Specification”, X/Open Preliminary Specification, July 1993