7. LAYER 5 - SESSION LAYER

Table Of Contents:

7.1 Synchronization **7-2**
7.2 Remote Procedure Calls **7-4**

Layer 5 is basically an invention of the ISO OSI committee. It does not add many communications features/functionality, and is thus termed a very "thin" layer. On many systems, the Layer 5 features are disabled, but you should nonetheless know what failures can be prevented by a session layer.

I will also briefly discuss Remote Procedure Calls, which sometimes are implemented with a connectionless session.

7.1 Synchronization

The main function of Layer 5 is to provide a way for the session users to establish connections, called Sessions, and transfer data (possibly using half-duplex) over them in a safe and orderly manner, even if the end computers fail and re-boot. This extra layer is necessary because, though the transport layer can recover from network layer intermediate node failures, it cannot always recover from some problems if an end computer fails and re-boots.

Here is the problem. Say you are transferring a database file from one end computer to another. A TPDU gets to the destination transport layer, and an acknowledgement is send back. Unfortunately, this can happen before the destination application has written the received data to disk. If the destination transport layer reboots, it will receive a restart request from the source, but since the source thinks it knows how much data has been received at the destination it will not bother to resend it. Unfortunately, one TPDU will be lost as it was caught in RAM as the destination computer failed.

The problem is that the two end applications do not hand shake that the data sent got 'completely' handled by the destination application. This is call synchronization.

Interestingly, in semester 96-3, a Cmpt 371 student noticed that he could have several clients call his server sequentially, and each completely finish and die, yet he had not told his server which file to put the second caller's data in! This means that the server was not only queuing calls, it was queuing incoming data. If the caller had sent and had its data acknowledged, and was designed to then do an abortive disconnect, this would allow the server to later get data from its transport layer and handle the data. But if the server failed with this data queued in RAM, the client application/user thought that the data got thru but the server application never received it up from the underlying transport process!

The session layer provides a a facility to log data to disk until the destination application has acknowledged (with a procedure call?) that it has completely handled that hunk of data. This allows rollback and recovery if an end node fails.

7.2 Remote Procedure Calls

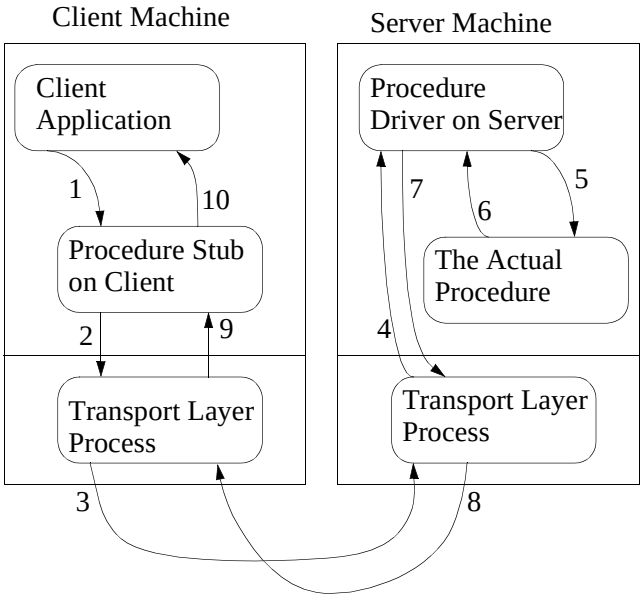
Remote procedure calls try to give a programmer the illusion that a program on one machine can call a procedure located on another machine. This requires that the calling program packetize up the parameters to send to the destination, and indicates which procedure is to be called at the destination machine. Generally, there is a layer 4 well-known port number monitored by a server which handles incoming procedure calls.

The server cannot normally start up an application containing the destination procedure, and call the procedure in that program (as that program is already running). Thus the destination procedure is usually in a dynamic link library that the RPC server can dynamically link to and call.

RPC does not fit into the OSI reference model very well, as it is designed to be fast and is not multi-layered. But some people have suggested that an RPC can be thought of as a short session. If implemented as a session, normally a connectionless session layer protocol is used.

When a client calls a remote procedure, the call is actually made to a local stub (procedure). The stub marshalls parameters together by value into a message, and transmits a request message to the server.

The Ten Steps of an Remote Procedure Call



When the RPC session message arrives, parameters are "unmarshalled" by a driver and the server procedure is called (i.e. 'driven').

The procedure result traces the same path in the opposite direction.

A key design issue in RPC systems is transparency. The idea is to make this process as transparent as possible to the client programmer, so she doesn't have to be concerned with the communications details.

The main problem in implementating transparency occurs with parameter passing. If call-by-reference allowed, the reference (i.e. pointer, e.g. $FF34A6D3_{16}$) means nothing at the server end.

Possible solutions:

- system replaces call-by-reference with call-by-copy/ return-changed-copy (i.e. a copy of the actual parameter is sent to server stub, which then gives a local pointer to the parameter to remote procedure). This can fail under certain pathological conditions. The following example is taken from Figure 7-14 of a previous edition [Tannenbaum 88] of your textbook.


```
Pascal PROGRAM MAIN (output);
VAR a:INTEGER;

PROCEDURE INCREMENT_EACH (VAR X,Y:INTEGER);
BEGIN
    X:=X+1;
    Y:=Y+1;
END;

BEGIN (*MAIN*)
    a:=0;
    INCREMENT_EACH(a,a);
    Writeln(a);
END.
```

You would expect this to increment 'a' by 2, but when the procedure is remote, 'a' is only incremented by 1 since each copy of the parameter is incremented separately.

- Alternatively, you could adopt a policy prohibiting the use of pointers, etc. But then the transparency of letting the client applications programmer not care about whether the procedure is remote or not is gone as rules for local and remote procedures are now different.

Finally, realize that for romote procedure calls, dynamic link libraries, and distributed objects (e.g. CORBA, Java RMI, Microsoft DCOM/OLE/ ACTIVEX), multiple programs could be calling at the same time, and critical sections need protecting!

7.3 References

[Tanenbaum88] "Computer Networks, 2nd ed." by Andrew Tanenbaum, Prentice-Hall, 1988,