Session Layer

The session layer resides above the transport layer, and provides "value added" services to the underlying transport layer services. The session layer (along with the presentation layer) add services to the transport layer that are likely to be of use to applications, so that each application doesn't have to provide its own implementation.

It is the thinnest layer in the OSI model. At the time the model was formulated, it was not clear that a session layer was needed.

The session layer provides the following services:

Dialog management: Deciding whose turn it is to talk. Some applications operate in half-duplex mode, whereby the two sides alternate between sending and receiving messages, and never send data simultaneously.

In the ISO protocols, dialog management is implemented through the use of a *data token*. The token is sent back and forth, and a user may transmit only when it possesses the token.

Synchronization: Move the two session entities into a known state.

The transport layer handles only communication errors, synchronization deals with upper layer errors. In a file transfer, for instance, the transport layer might deliver data correctly, but the application layer might be unable to write the file because the file system is full.

Users can split the data stream into *pages*, inserting *synchronization points* between each page. When an error occurs, the receiver can *resynchronize* the state of the session to a previous synchronization point. This requires that the sender hold data as long as may be needed.

Synchronization is achieved through the use of sequence numbers. The ISO protocols provide both *major* and *minor* synchronization points. When resynchronizing, one can only go back as far as the previous major synchronization point. In addition, major synchronization points are acknowledged through explicit messages (making their use expensive). In contrast, minor synchronization points are just markers.

Activity management: Allow the user to delimit data into logical units called *activities*. Each activity is independent of activities that come before and after it, and an activity can be processed on its own.

Activities might be used to delimit files of a multi-file transfer.

Activities are also used for *quarantining*, collecting all the messages of a multi-message exchange together before processing them. The receiving application would begin processing messages only after all the messages had arrived. This provides a way of helping insure that all or none of a set of operations are performed.

For example, a bank transaction may consist of locking a record, updating a value, and then unlocking the record. If an application processed the first operation, but never received the remaining operations (due to client or network failures), the record would remain locked forever. Quarantining addresses this problem.

Exception handling: A General purpose mechanism for reporting errors.

Note: The TCP/IP protocols do not include a session layer at all.

Remote Procedure Call (RPC)

Remote Procedure Call (RPC) provides a different paradigm for accessing network services. Instead of accessing remote services by sending and receiving messages, a client invokes services by making a local procedure call. The local procedure hides the details of the network communication.

When making a remote procedure call:

1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.
2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.

The main goal of RPC is to hide the existence of the network from a program. As a result, RPC doesn't quite fit into the OSI model:

1. The message-passing nature of network communication is hidden from the user. The user doesn't first open a connection, read and write data, and then close the connection. Indeed, a client often doesn not even know they are using the network!
2. RPC often omits many of the protocol layers to improve performance. Even a small performance improvement is important because a program may invoke RPCs often. For example, on (diskless) Sun workstations, every file access is made via an RPC.

RPC is especially well suited for client-server (e.g., query-response) interaction in which the flow of control alternates between the caller and callee. Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again. The following steps take place during an RPC:

1. A client invokes a *client stub* procedure, passing parameters in the usual way. The client stub resides within the client's own address space.
2. The client stub *marshalls* the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.
3. The client stub passes the message to the transport layer, which sends it to the remote server machine.
4. On the server, the transport layer passes the message to a *server stub*, which demarshalls the parameters and calls the desired server routine using the regular procedure call mechanism.
5. When the server procedure completes, it returns to the server stub (e.g., via a normal procedure call return), which marshalls the return values into a message. The server stub then hands the message to the transport layer.
6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.
7. The client stub demarshalls the return parameters and execution returns to the caller.

RPC Issues

Issues that must be addressed:

Marshalling: Parameters must be *marshalled* into a standard representation.

Parameters consist of simple types (e.g., integers) and compound types (e.g., C structures or Pascal records). Moreover, because each type has its own representation, the types of the various parameters must be known to the modules that actually do the conversion. For example, 4 bytes of characters would be uninterpreted, while a 4-byte integer may need to the order of its bytes reversed.

Semantics: *Call-by-reference* not possible: the client and server don't share an address space. That is, addresses referenced by the server correspond to data residing in the client's address space.

One approach is to simulate call-by-reference using *copy-restore*. In copy-restore, call-by-reference parameters are handled by sending a copy of the referenced data structure to the server, and on return replacing the client's copy with that modified by the server.

However, copy-restore doesn't work in all cases. For instance, if the same argument is passed twice, two copies will be made, and references through one parameter only changes one of the copies.

Binding: How does the client know *who* to call, and *where* the service resides?

The most flexible solution is to use *dynamic binding* and find the server at run time when the RPC is first made. The first time the client stub is invoked, it contacts a name server to determine the transport address at which the server resides.

Transport protocol: What transport protocol should be used?

Exception handling: How are errors handled?

Binding

We'll examine one solution to the above issues by considering the approach taken by Birrell and Nelson [**?**]. Binding consists of two parts:

Naming refers to what service the client wants to use.

> In B&N, remote procedures are named through *interfaces*. An interface uniquely identifies a particular service, describing the types and numbers of its arguments. It is similar in purpose to a type definition in programming languauges.
>
> For example, a "phone" service interface might specify a single string argument that returns a character string phone number.

Locating refers to finding the transport address at which the server actually resides. Once we have the transport address of the service, we can send messages directly to the server.

In B&N's system, a server having a service to offer *exports* an interface for it. Exporting an interface registers it with the system so that clients can use it.

A client must *import* an (exported) interface before communication can begin. The export and import operations are analogous to those found in object-oriented systems.

Interface names consists of two parts:

1. A unique *type* specifies the interface (service) provided. Type is a high-level specification, such as "mail" or "file access".
2. An *instance* specifies a particular server offering a type (e.g., "file access on *wpi*").

Name Server

B&N's RPC system was developed as part of a distributed system called Grapevine. Grapevine was developed at Xerox by the same research group the developed the Ethernet.

Among other things, Grapevine provides a distributed, replicated database, implemented by servers residing at various locations around the internet.

Clients can query, add new entries or modify existing entries in the database.

The Grapevine database maps character string keys to entries called *RNames*. There are two types of entries:

Individual: A single instance of a service. Each server registers the transport address at which its service can be accessed and every instance of an interface is registered as an individual entry. Individual entries map instances to their corresponding transport addresses.

Group: The type of an interface, which consists of a list of individual *RNames*. Group entries contain RNames that point to servers providing the service having that group name. Group entries map a type (interface) to a set of individual entries providing that service.

For example, if *wpi* and *bigboote* both offered file access, the group entry "file access" would consists of two individual RNames, one for *wpi* and *bigboote*'s servers.

Service Interface Export/Import

When a server wishes to export an interface:

1. It calls its server stub, which then calls Grapevine, passing it the type and instance of the service it wishes to register. Once the interface has been registered with Grapevine, it can be imported by the client. Note: Grapevine insures that both an individual and a group entry has been established for the exported service.

2. The server stub then records information about the instance in an internal *export table*.

   In B&N, there is one export table per machine, containing entries for all currently exported interfaces. This table is used to map incoming RPC request messages to their corresponding server procedure.

3. Each entry in the export table contains:

   a unique identifier that identifies that interface, and

   a pointer to the server stub that should be called to invoke the interface service. The unique identifier is never reused. If the server crashes and restarts, new identifiers are used.

The client binds to an exported service as follows:

1. The client stub calls the Grapevine database to find an instance of the desired type.

2. The database server returns the desired group entry, and the client chooses one of the individual servers.

3. The client stub sends a message to the selected server stub asking for information about that instance.

4. The server stub returns the unique identifier and index of the appropriate export table entry. The client saves the [identifier, index] pair as it will need it when actually making an RPC.

5. The client is now ready to actually call the remote procedure:

   (a) The client sends the export table index and the unique identifier together with the parameters of the call to the server.

   (b) Upon receipt of a message, the server stub uses the table index contained in the message to find the appropriate entry in its export table.

(c) The server stub then compares the provided identifier with the one in the table. If they differ, reject the procedure call as invalid.

Otherwise call the server stub procedure.

Binding Notes

Note: The unique identifiers in the export table change whenever a server crashes and restarts, allowing the client to detect server restarts between calls. In those cases where a client doesn't care if the server has restarted, it simply rebinds to another instance of the interface and restarts the remote call.

Note: Identifiers are managed by the server and are not stored in the Grapevine database. Storing them in the Grapevine database would reduce the number of messages exchanged during the binding phase. However, the current approach greatly reduces the load on the Grapevine servers. In most cases, when a server exports an interface to Grapevine, the entry will have been registered previously, and no updates to the database are required (updates are expensive because the database is distributed).

Using Grapevine's database provides *late binding*. Binding callers to specific servers at runtime makes it possible to move the server to another machine without requiring changes to client software.

Finally, the separate registering of types and instances provides great flexibility. Rather than binding to a specific instance, a client asks for a specific type. Because all instances of a type implement the same interface, the RPC support routines would take the list of instances returned by Grapevine and chose the one that is closest to the client.

How is binding done on other systems?

Semantics of RPC

Unlike normal procedure calls, many things can go wrong with RPC. Normally, a client will send a request, the server will execute the request and then return a response to the client. What are appropriate semantics for server or network failures? Possibilities:

1. Just hang forever waiting for the reply that will never come. This models regular procedure call. If a normal procedure goes into an infinite loop, the caller never finds out. Of course, few users will like such semantics.
2. Time out and raise an exception or report failure to the client. Of course, finding an appropriate timer value is difficult. If the remote procedure takes a long time to execute, a timer might time-out too quickly.
3. Time out and retransmit the request.

While the last possibility seems the most reasonable, it may lead to problems. Suppose that:

1. The client transmits a request, the server executes it, but then crashes before sending a response. If we don't get a response, is there any way of knowing whether the server acted on the request?
2. The server restarts, and the client retransmits the request. What happens? Now, the server will reject the retransmission because the supplied unique identifier no longer matches that in the server's export table. At this point, the client can decide to rebind to a new server and retry, or it can give up.
3. Suppose the client rebinds to the another server, retransmits the request, and gets a response. How many times will the request have been executed? At least once, and possibly twice. We have no way of knowing.

Operations that can safely be executed twice are called *idempotent*. For example, fetching the current time and date, or retrieving a particular page of a file.

Is deducting $10,000 from an account idempotent? No. One can only deduct the money once. Likewise, deleting a file is not idempotent. If the delete request is executed twice, the first attempt will be successful, while the second attempt produces a "nonexistent file" error.

RPC Semantics

While implementing RPC, B&N determined that the semantics of RPCs could be categorized in various ways:

Exactly once: The most desirable kind of semantics, where every call is carried out exactly once, no more and no less. Unfortunately, such semantics cannot be achieved at low cost; if the client transmits a request, and the server crashes, the client has no way of knowing whether the server had received and processed the request before crashing.

At most once: When control returns to the caller, the operation will have been executed no more than once. What happens if the server crashes? If the server crashes, the client will be notified of the error, but will have no way of knowing whether or not the operation was performed.

At least once: The client just keeps retransmitting the request until it gets the desired response. On return to the caller, the operation will have be performed at least one time, but possibly multiple times.

Transport Protocols for RPC

Can we implement RPC on top of an existing transport protocol such as TCP? Yes. However, reliable stream protocols are designed for a different purpose: high throughput. The cost of setting up and terminating a connection is insignificant in comparison to the amount of data exchanged. Most of the elapsed time is spent sending data.

With RPC, low latency is more important than high throughput. If applications are going to use RPC much like they use regular procedures (e.g., over and over again), performance is crucial.

RPC can be characterized as a specific instance of *transaction-oriented communication*, where:

- A transaction consists of a single request and a single response.
- A transaction is initiated when a client sends a request and terminated by the server's response.

How many TCP packets would be required for a single request-response transaction? A minimum of 5 packets: 3 for the initial handshake, plus 2 for the FIN and FIN ACK (assuming that we can piggy back data and a FIN on the third packet of the 3-way handshake).

A transaction-oriented transport protocol should efficiently handle the following cases:

1. Transactions in which both the request and response messages fit in a single packet. The response can serve as an acknowledgment, and the client handles the case of lost packets by retransmitting the original request.
2. Large multi-packet request and response messages, where the data does not necessarily fit in a single packet. For instance, some systems use RPC to fetch pages of a file from a file server. A single-packet request would specify the file name, the starting position of the data desired, and the number of bytes to be read. The response may consist of several pages (e.g. 8K bytes) of data.