- Reliable client-server communication

In many cases, fault tolerance in distributed systems concentrates on faulty processes. However, we also need to consider communication failures. Most of the failure models discussed previously apply equally well to communication channels. In particular, a communication channel may exhibit crash, omission, timing, and arbitrary failures. In practice, when building reliable communication channels, the focus is on masking crashes and omission failures. Arbitrary failures may occur in the form of duplicate messages, resulting from the fact that in a computer network messages may be buffered for a relatively long time, and are reinjected into the network after the original sender has already issued a retransmission.

Point-to-Point Communication

In many distributed systems, reliable point-to-point communication is established by making use of a reliable transport protocol, such as TCP. TCP masks omission failures, which occur in the form of lost messages, by using acknowledgments and retransmissions. Such failures are completely hidden from a TCP client.

However, crash failures of connections are not masked. A crash failure may occur when (for whatever reason) a TCP connection is abruptly broken so that no more messages can be transmitted through the channel. In most cases, the client is informed that the channel has crashed by raising an exception. The only way to mask such failures is to let the distributed system attempt to automatically set up a new connection, by simply resending a connection request. The underlying assumptions that the other side is still, or again, responsive to such requests.

RPC Semantics in the Presence of Failures

Let us now take a closer look at client-server communication when using high-level communication facilities such as Remote Procedure Calls (RPCs). The goal of RPC is to hide communication by making remote procedure calls look just like local ones. With a few exceptions, so far we have come fairly close. Indeed, as long as both client and server are functioning perfectly, RPC does its job well.

The problem comes about when errors occur. It is then that the differences between local and remote calls are not always easy to mask.

To structure our discussion, let us distinguish between five different classes of failures that can occur in RPC systems, as follows:

1. The client is unable to locate the server.

2. The request message from the client to the server is lost.

3. The server crashes after receiving a request.

4. The reply message from the server to the client is lost.

5. The client crashes after sending a request.

Each of these categories poses different problems and requires different solutions.

Client Cannot Locate the Server

To start with, it can happen that the client cannot locate a suitable server. All servers might be down, for example. Alternatively, suppose that the client is compiled using a particular version of the client stub, and the binary is not used for a considerable period of time. In the meantime, the server evolves and a new version of the interface is installed; new stubs are generated and put into use. When the client is eventually run, the binder will be unable to match it up with a server and will report failure. While this mechanism is used to protect the client from accidentally trying to talk to a server that may not agree with it in terms of what parameters are required or what it is supposed to do, the problem remains of how this failure should be dealt with.

One possible solution is to have the error raise an exception. In some languages, (e.g., Java), programmers can write special procedures that are invoked upon specific errors, such as division by zero. In C, signal handlers can be used for this purpose. In other words, we could define a new signal type SIGNO SERVER, and allow it to be handled in the same way as other signals.

This approach, too, has drawbacks. To start with, not every language has exceptions or signals. Another point is that having to write an exception or signal handler destroys the transparency we have been trying to achieve. Suppose that you are a programmer and your boss tells you to write the sum procedure. You smile and tell her it will be written, tested, and documented in five minutes. Then she mentions that you also have to write an exception handler as well, just in case the procedure is not there today. At this point it is pretty hard to maintain the illusion that remote procedures are no different from local ones, since writing an exception handler for "Cannot locate server" would be a rather unusual request in a single-processor system. So much for transparency.

Lost Request Messages

The second item on the list is dealing with lost request messages. This is the easiest one to deal with: just have the operating system or client stub start a timer when sending the request. If the timer expires before a reply or acknowledgment comes back, the message is sent again. If the message was truly lost, the server will not be able to tell the difference between the retransmission and the original, and everything will work fine. Unless, of course, so many request messages are lost that the client gives up and falsely concludes that the server is down, in which case we are back to "Cannot locate server." If the request was not lost, the only thing we need to do is let the server be able to detect it is dealing with a retransmission. Unfortunately, doing so is not so simple, as we explain when discussing lost replies.

Server Crashes

The next failure on the list is a server crash. The normal sequence of events at a server is shown in Fig. 8-7(a). A request arrives, is carried out, and a reply is sent. Now consider Fig. 8-7(b). A request arrives and is carried out, just as before, but the server crashes before it can send the reply. Finally, look at Fig. 8-7(c). Again a request arrives, but this time the server crashes before it can even be carried out. And, of course, no reply is sent back.

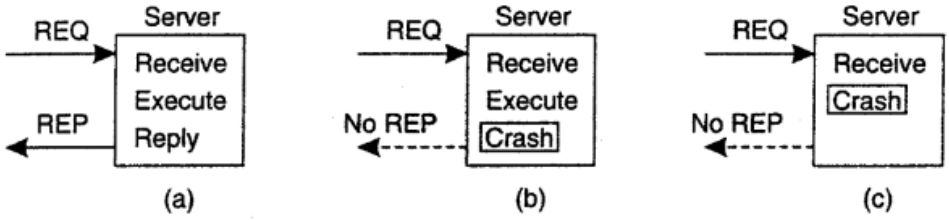

Figure 8-7. A server in client-server communication. (a) The normal case. (b) Crash after execution. (c) Crash before execution.

The annoying part of Fig. 8-7 is that the correct treatment differs for (b) and (c). In (b) the system has to report failure back to the client (e.g., raise, an exception), whereas in (c) it can just retransmit the request. The problem is that the client's operating system cannot tell which is which. All it knows is that its timer has expired.

Assume that the server crashes and subsequently recovers. It announces to all clients that it has just crashed but is now up and running again. The problem is that the client does not know whether its request to print some text will actually be carried out.

There are four strategies the client can follow. First, the client can decide to never reissue a request, at the risk that the text will not be printed. Second, it can decide to always reissue a request, but this may lead to its text being printed twice. Third, it can decide to reissue a request only if it did not yet receive an acknowledgment that its print request had been delivered to the server. In that case, the client is counting on the fact that the server crashed before the print request could be delivered. The fourth and last strategy is to reissue a request only if it has received an acknowledgment for the print request.

With two strategies for the server, and four for the client, there are a total of eight combinations to consider. Unfortunately, no combination is satisfactory. To explain, note that there are three events that can happen at the server: send the completion message (M), print the text (P), and crash (C). These events can occur in six different orderings:

1. M ~P ~C: A crash occurs after sending the completion message and printing the text.

2. M ~C (~P): A crash happens after sending the completion message, but before the text could be printed.

3. p ~M ~C: A crash occurs after sending the completion message and printing the text.

4. P~C(~M): The text printed, after which a crash occurs before the completion message could be sent.

5. C (~P ~M): A crash happens before the server could do anything.

6. C(~M ~P): A crash happens before the server could do anything.

The parentheses indicate an event that can no longer happen because the server already crashed. Fig. 8-8 shows all possible combinations. As can be readily verified, there is no combination of client strategy and server strategy that will work correctly under all possible event sequences. The bottom line is that the client can never know whether the server crashed just before or after having the text printed.

Client	Server					
	Strategy M → P			Strategy P → M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

OK = Text is printed once
DUP = Text is printed twice
ZERO = Text is not printed at all

Figure 8-8. Different combinations of client and server strategies in the presence of server crashes.

In short, the possibility of server crashes radically changes the nature of RPC and clearly distinguishes single-processor systems from distributed systems. In the former case, a server crash also implies a client crash, so recovery is neither possible nor necessary. In the latter it is both possible and necessary to take action.

Lost Reply Messages

Lost replies can also be difficult to deal with. The obvious solution is just to rely on a timer again that has been set by the client's operating system. If no reply is forthcoming within a reasonable period, just send the request once more. The trouble with this solution is that the client is not really sure why there was no answer. Did the request or reply get lost, or is the server merely slow? It may make a difference.

In particular, some operations can safely be repeated as often as necessary with no damage being done. A request such as asking for the first 1024 bytes of a file has no side effects and can be executed as often as necessary without any harm being done. A request that has this property is said to be idempotent.

Now consider a request to a banking server asking to transfer a million dollars from one account to another. If the request arrives and is carried out, but the reply is lost, the client will not know this and will retransmit the message. The bank server will interpret this request as a new one, and will carry it out too. Two million dollars will be transferred. Heaven forbid that the reply is lost 10 times. Transferring money is not idempotent.

One way of solving this problem is to try to structure all the requests in an idempotent way. In practice, however, many requests (e.g., transferring money) are inherently non idempotent, so something else is needed. Another method is to have the client assign each request a sequence number. By having the server keep track of the most recently received sequence number from each client that is using it, the server can tell the difference between an original request and a retransmission and can refuse to carry out any request a second time. However, the server will still have to send a response to the client. Note that this approach does require that the server maintains administration on each client. Furthermore, it is not clear how long to maintain this administration. An additional safeguard is to have a bit in the message header that is used to distinguish initial requests from retransmissions.

Client Crashes

The final item on the list of failures is the client crash. What happens if a client sends a request to a server to do some work and crashes before the server replies? At this point a computation is active and no parent is waiting for the result. Such an unwanted computation is called an orphan.

Orphans can cause a variety of problems that can interfere with normal operation of the system. As a bare minimum, they waste CPU cycles. They can also lock files or otherwise tie up valuable resources. Finally, if the client reboots and does the RPC again, but the reply from the orphan comes back immediately afterward, confusion can result.

What can be done about orphans? Nelson (1981) proposed four solutions. In solution 1, before a client stub sends an RPC message, it makes a log entry telling what it is about to do. The log is kept on disk or some other medium that survives crashes. After a reboot, the log is checked and the orphan is explicitly killed off.

This solution is called orphan extermination.

The disadvantage of this scheme is the horrendous expense of writing a disk record for every RPC. Furthermore, it may not even work, since orphans themselves may do RPCs, thus creating grand orphans or further descendants that are difficult or impossible to locate. Finally, the network may be partitioned, due to a failed gateway, making it impossible to kill them, even if they can be located. All in all, this is not a promising approach.

In solution 2. called reincarnation, all these problems can be solved without the need to write disk records. The way it works is to divide time up into sequentially numbered epochs. When a client reboots, it broadcasts a message to all machines declaring the start of a new epoch. When such a broadcast comes in, all remote computations on behalf of that client are killed. Of course,

if the network is partitioned, some orphans may survive. Fortunately, however, when they report back, their replies will contain an obsolete epoch number, making them easy to detect.

Solution 3 is a variant on this idea, but somewhat less draconian. It is called gentle reincarnation. When an epoch broadcast comes in, each machine checks to see if it has any remote computations running locally, and if so, tries its best to locate their owners. Only if the owners cannot be located anywhere is the computation killed.

Finally, we have solution 4, expiration, in which each RPC is given a standard amount of time, T, to do the job. If it cannot finish, it must explicitly ask for another quantum, which is a nuisance. On the other hand, if after a crash the client waits a time T before rebooting, all orphans are sure to be gone. The problem to be solved here is choosing a reasonable value of Tin the face of RPCs with wildly differing requirements.