Interprocess Communication*

Sape J. Mullender

One of the fundamental characteristics of a distributed system is that it has multiple processing elements which may fail independently. Interprocess communication mechanisms are not merely there to allow the processes in a distributed system to communicate; they are also there to provide a mechanism that shields one process from failures of another. A vital function of communication mechanisms is the prevention of crashes in one process bringing down another.

Interprocess communication mechanisms provide a small and comprehensive set of interaction possibilities between processes which, if used exclusively for the interaction between them, forces clean and simple interfaces between processes.

In centralized multiprocessing systems, processes can communicate via shared memory, or Unix pipes, or even RAM files. There is an advantage in making use of the same mechanisms for interaction between processes on one processor, a shared-memory multiprocessor, and a distributed system. It allows redistribution of processes over processors and porting multiprocess applications between centralized and distributed systems.

Interprocess communication mechanisms thus serve four important functions in distributed systems.

1. They allow communication between separate processes over a computer network,
2. they provide firewalls against failures, and provide the means to cross protection boundaries,
3. they enforce clean and simple interfaces, thus providing a natural aid for modular structuring of large distributed applications, and
4. they hide the distinction between local and remote communication, thus allowing static or dynamic reconfiguration.

Interprocess communication mechanisms must be as unobtrusive as possible. This means that they must be easy to use and that they provide the performance that

allows them to be used anywhere. One of the easiest and simplest communication interfaces is *remote procedure call*. It has already become one of the most popular in distributed systems research and appears to make its way slowly into commercial distributed and networked systems. Later in this chapter, we shall look in detail at how remote procedure call works.

When machine or protection boundaries must be crossed, interprocess communication mechanisms necessarily incur the performance penalties of operating system interfaces, the hardware mechanisms that need to be invoked — network and network interfaces, memory management —, protocol machinery that deals with host, network and process failures, and the inefficiencies associated with making interprocess communication interfaces uniform.

Interprocess communication services should deliver information with *minimum latency*, *maximum throughput*, and in the case of continuous media (audio and video) they should also *minimize jitter*, the irregularities in the latency. Interprocess communication should be *authenticated* (receiver knows who sends it) and *secure* (sender knows who receives it). When communicating over a network, the interprocess communication mechanism should also hide as many failures in the communication medium as possible. Finally, failures that cannot be hidden should be reported as accurately as the system allows.

Error recovery and failure detection play a fundamental rôle in interprocess communication and we shall often return to this theme throughout this chapter.

## 1 Computer Networks

A computer network delivers data between *nodes*. Nodes can be unprocessors or multiprocessors. We assume that nodes can *fail independently*. The interconnection between the processors of a shared-memory multiprocessor — usually a bus — connects entities that do not fail independently, so we shall not view it as a computer network.

Computer networks differ in the area they serve (a building, a campus, a city, a country, a continent), in the interconnection topology they offer (regular or irregular point-to-point topology, or a broadcast medium), in the transmission speeds, and the way in which data is packaged.

In practically all networks, there is a non-negligible probability that data received are different from those sent. Sometimes, just one bit is flipped, sometimes hundreds of bits in a row are lost. To catch such errors, *packets* of bits are transmitted and, attached to each packet, is redundancy that allows *error detection* and sometimes even *error correction*. 
Error detection requires less redundancy than error correction. Which is better depends on the error rate, the number of bits in error as a percentage of the total number sent. Usually, the error rate is very low (e.g., for local-area networks, one faulty bit in $10^{12}$ is not unusual) and then the overhead of error detection and retransmission is much less than that of an error-correcting code.

Data packets are typically between a hundred and a few thousand bytes in size and they have a checksum as error-detecting redundancy attached to the end. The checksum is computed using an algorithm that catches most "typical" errors and that can be computed in real time. Cyclic redundancy checksums (CRC) are most popular.

In some circumstances, however, an error-correcting code is the best solution. Communication between an deep-space probe and an earth station, for instance, has a high error rate, combined with a tremendous end-to-end latency (several hours) and then an error-correcting code works better than error detection and retransmission.

Another, closer to home example of data transmission where error detection and retransmission is not a good idea is in interactive continuous media. The maximum acceptable end-to-end latency in audio or audio/visual person-to-person interaction is a few tens of milliseconds. Over large distances, retransmission would just cost too much time --- in continuous media, late data is useless data. Fortunately, the human ear and eye have some error correction built in: If there are small gaps in the data stream, the ear or eye won't notice. In audio, the gap can be a few milliseconds, in video, a few tens of milliseconds. Audio and video are usually transmitted without any error detection or error correction at all.

As early as in 1974, Hasler AG, now part of ASCOM, used a register-insertion ring network called SILK that carried data in very small fixed-sized packets in order to use it for digital telephony. The idea never caught on at the time. Now, with the advent of fibre-optical networks for long-haul data transport, phone companies are looking for networks that can carry all types of data --- digital audio and video as well as data. As a result of this, networks that switch small fixed-size packets (called cells) are rapidly gaining popularity under the surprising name Asynchronous Transfer Mode (ATM) networks.

ATM networks carry cells of a fixed size with a small header (e.g., 5 bytes in Broadband ISDN (B-ISDN)) and 48 bytes of data. The header of a cell contains a virtual circuit identifier which is used to transport cells along previously created virtual circuits through the network. Individual cells do not have checksums on the data they carry, so a higher level of protocol has to detect damaged cells.

Error detection and, if necessary, error correction is done at the end points of the
virtual circuits. Here, cells are collected into larger data units, packets. In the header of each cell is a bit that can be used to indicate the end of a logical message. Several virtual circuits may lead to one host, so the host must demultiplex packets from different virtual circuits into the correct packet buffers. The layer responsible for the demultiplexing and the packet-level error detection (with a per-packet checksum) is the ATM Adaptation Layer (AAL). The AAL is used for data connections which have to be error free. Audio or video connections, which need no error detection (and thus no packetization) only make use of the demultiplexing function.

ATM has good properties for low-latency continuous media transmission. The access time to the network for high-priority traffic is one cell time. Also, the latency caused by the time it takes to fill up a cell is small. For telephone-grade audio, which uses 8-bit samples at a frequency of 8 KHz, one cell represents 6 ms of audio. The loss of a single such cell is barely detectable by the human ear.

Most networks today make use of physical connections, such as cables, wires, or optical fibres. Telephone companies also make use of microwave or satellite links, but they offer only point-to-point communication to customers. Wireless networks have been used in unusual environments. The University of Hawaii, for instance, which is scattered over a number of islands, already used the ALOHA packet radio network in the early seventies to interconnect pieces of the universities (Abramson [1970]).

But wireless communication will gain in importance now that fairly powerful portable workstations — laptop, notebook and palmtop computers — are becoming common. People will want to be connected to the rest of the world even when on the road. The next generation of cellular telephone technology will no doubt be digital and therefore be useable to connect travelling computer users to the network.

Since one antenna can only support a moderate aggregate bandwidth, high point-to-point bandwidth can only be realized by keeping the number of connections per antenna low. This implies that one antenna should only cover a small geographical area. Cellular telephone systems today use cells (not to be confused with ATM cells) of a few kilometres in diameter. High-bandwidth wireless computer networks for buildings (where the number of users per square metre is large) may use cells of only a few metres in diameter. Such small cells offer the additional benefit that the portable computer devices using it do not have to have powerful — and thus heavy — batteries.

Wireless networks, as well as broadcast cable networks (e.g., Ethernet), are particularly vulnerable to unauthorized eavesdropping, modification or insertion of network traffic. In such networks, especially, authentication and privacy must be
ensured by the use of encryption.

The weak spot in most computer networks is where they connect to the computer. The design of today's network interfaces appears to be inspired primarily by that of disk controllers: linked lists of I/O descriptors that indicate buffers for receiving (reading) or sending (writing) using DMA. The controller works down the list doing the indicated work. For network controllers, two such queues are usually maintained, one for sending and one for receiving.

Packets may arrive for several receiving processes in one host, and it would be desirable to receive data directly in the receiving process' address space. Most network controllers put received packets in the first available buffer on the queue, without interpreting the packet contents. The demultiplexing of the incoming packet stream, therefore, has to be done by protocol software and copying packet data to the appropriate location in the receiving process' address space is usually unavoidable.

In networks with bandwidths around 10 Mbps and 10 MIPS processors the latency increases by around 10%. If network bandwidths become ten times higher, the penalty would increase to between 50% and 100% (Schroeder and Burrows [1989]).

With the advent of ATM networks, host interfaces will have to start doing the demultiplexing of the incoming cell stream. The cells arrive at a rate where the demultiplexing cannot be done realistically in software. For operating system designers this is good news, because now it will be possible to program any ATM host interface of reasonable design to receive data from each virtual circuit into its own set of buffers. These buffers can then be allocated directly in the desired location in the receiving process' address space.

2 Protocol Organization

The physical medium is almost never error free. It corrupts bits, sometimes to the extent that packet boundaries are no longer recognized. Packets usually have a checksum that allows packets with bit errors in them to be detected and discarded. Checksums allow us to view the physical medium as one where packets are occasionally lost, or where communication may fail altogether, but where no other errors occur.

Media errors can be corrected by what the ISO OSI model calls a datalink-layer protocol. Such a protocol uses timeout, acknowledgements (acks) and retransmission to detect and correct packet loss. As a consequence, packets are not always
2 Protocol Organization

received in the same order as the one in which they were sent. The data link layer may or may not restore order before delivering packets to the next layer. Order is restored, however, at the cost of a higher latency — correctly received packets may have to wait for the delivery of earlier packets that need to be retransmitted.

In point-to-point networks, packets may have to travel several hops before reaching their destination. Network-layer protocols, are responsible for routing packets through the network. They can make routing decisions for every individual packet, so that packets are delivered independently of other packets. We call this a datagram or connectionless service. They can also make routing decisions once for a long string of packets. All packets in a connection then follow the same route through the network and it is then a simple matter to detect and correct packet loss and maintain packet order within the connection. We call this a virtual-circuit or connection-oriented service. ATM networks are always based on virtual-circuits because packets arrive to rapidly to make routing decisions on a per-packet basis.

A connectionless network-layer service never guarantees that packets are delivered in the order sent. If it gives a reliable service, it will depend on the underlying datalink-layer protocols to do so. It is unlikely, though, that failures of intermediate network nodes are masked to the extent that no packets are ever lost.

A connection-oriented network-layer service usually does guarantee that packets are delivered in the order sent and that no packets are lost, but not always, as, for instance, in the case of ATM. Here also, intermediate-node failures are usually not masked. Virtual circuits break when such a failure occurs and it is up to higher layers of protocol to create another virtual circuit that avoids the crashed intermediate node.

Network-layer protocols deliver data between hosts. The real senders and recipients of data, however, are usually processes running on those hosts, so many independent data streams may exist between hosts. The protocol that is responsible for delivering data end-to-end between processes in the network is the Transport Layer Protocol. If the combination of the services of the underlying protocols — Physical, Data Link and Network — does not provide the desired reliability, then the Transport Layer Protocols must do it.

Transport protocols may run over connection-oriented or connectionless network layer services. One would expect that a transport protocol running over an underlying connection-oriented service would be simpler, because much of the work is done in lower-level protocols, but this is not really the case. Network connections may break and then the transport protocol must create new ones. In ATM networks, cells are only assembled into packets at the communication endpoints, so it is only there that transmission errors can be detected and corrected.
Transport protocols can offer a variety of services. The most common are a reliable stream of packets, a reliable byte stream, or a reliable stream of logical messages. Some transport protocols offer reliable pairs of request and reply messages for use in remote operations or remote procedure call.

3 Fundamental Properties of Protocols

We have seen that networks exhibit failures that manifest themselves as lost packets. As long as there is some communication — that is, not all packets are lost — these failures can be corrected using feedback in the form of acknowledgements and timeouts. When all packets are lost, such as when a cable breaks or a connector becomes unstuck, hosts can become disconnected altogether. If the processes that form the communication endpoints can crash, communication protocols cannot be made completely reliable.

If a client\footnote{I will use the terms client and server here to distinguish the two communicating processes. The discussion here applies equally well to processes communicating in different roles.} process sends a request message to a remote server process via a network that can lose packets, and, in spite of numerous retransmissions, no response ever comes back from the server, then the network may have broken, or the server may have crashed. As long as the fault is not repaired, the client has no way of finding out what happened.

The client cannot distinguish a server that went down from one that has become disconnected. By itself, this is no great problem; the real problem here is that the client has no way of finding out whether the server has received the request and started carrying out the requested work or not. As shown in Figure 1, it is no help to the client whether the server sends an ack upon reception of the request. The server may crash before it has a chance to start the actual work, or the network may break while the server is doing the work so that the reply cannot be sent back.

The client will be left in the same sort of uncertainty if the network were completely reliable. The server could crash just before it starts the work, or just after it finishes, but before it can communicate this fact. When communication is reliable, but processes are not, at least clients know that a server crash has occurred when no reply comes back. This uncertainty is easier to bear than that of not knowing whether the server has crashed or the network has become disconnected.

When a server comes back up after a crash, its client can resume communication with it by retransmitting the last — unanswered — request. But the server may have carried out that request once already. If the server remembers this fact, it
could recompute the reply and send that to the client. But if the server suffers total
amnesia in a crash and forgets what it did when it crashed, the server will not be able
to distinguish between new requests and requests that have already been carried out.
Knowing that the request is a retransmission does not really help. It can only serve
to warn a server that it may have carried out the request in a previous incarnation.

In this paper, we discuss protocols where crashes are always *amnesia failures.*
Protocols also exist that use *stable storage* to store information on that must survive
crashes. In an amnesia failure, a process suddenly stops and forgets all its state. It
then resumes operation from the initial state — it reboots. This failure model repres-
ts a large class of failures of processors and processes in actual systems. Within
this model, *exactly-once* message delivery, a desirable property of communication
protocols, cannot be achieved. One can aim for either *at-least-once* or *at-most-once*
message delivery.

At-least-once protocols deliver messages exactly once in the absence of failures,
but may deliver messages more than once when failures occur. Such protocols work
when requests are *idempotent,* that is, when carrying them out once has the same
effect as carrying them out several times. Most operations on files are idempotent:
Reading a block once has the same effect as reading it three times and the same is
true for writing. Creating or deleting files is almost idempotent: The first try will do
the work, subsequent tries will fail. SUN's Network File System (SUN NFS) uses
an at-least-once protocol (SUN RPC) for its file operations and, for that purpose, this is perfectly reasonable. NFS servers are, for purposes of the communication protocol, stateless, so there is no state to forget when the server crashes.

At-least-once message delivery hides communication failures and server failures and this only works in a very restricted set of circumstances. In most interactions between processes, losing the communication state is a serious failure and, although we have seen that it cannot be prevented, it certainly should not go unnoticed. At-most-once message delivery protocols detect the fact that the network, or one of the communicating processes has failed and report this fact.

At-most-once protocols operate in the context of a session — an association between two processes during which both maintain protocol state. Whenever state is lost, either as a consequence of a crash, a network failure, or because the state was deliberately discarded, the session is terminated. The communicating parties must agree on what is the current session, so that no messages sent in one session are received in another. To do this, sessions have unique names.

Generating unique names for sessions is not quite as easy as it appears. Remember that, when processes crash, they forget everything, including the name of the current session. If only one party crashes, the other can choose a new session identifier, but if both crash, this doesn’t work.

A good way to choose session identifiers is to use timestamps as part of the identifier. Most modern processors have a battery-operated clock whose value survives crashes. Even if they don’t, they can ask a network time service for the time when they come up. Another way is to use random numbers and hope that the random number generation is good enough to make the probability of re-using session identifiers sufficiently small.

Session identifiers must belong to an association of two processes. Some distributed systems use a form of functional addressing where a client sends a request for a remote operation to some generic service. An association between a client process and some generic service will not do, unless the service is stateless. Figure 2 illustrates what can happen if functional addressing is used injudiciously. Here, a client process sends a request to an abstract service which happens to be a replicated one with two server processes. The request goes to one of them and is carried out there. The response is lost, for instance, because the server crashes, and the client retransmits. The second server receives the retransmission and carries the request out for the second time, clearly violating the at-most-once principle.

Amoeba (Mullender et al. [1990]; Tanenbaum et al. [1990]) is an example of a distributed system using functional addressing. Client applications send requests to a service rather than to a specific server process. The name of a service is called
3 Fundamental Properties of Protocols

Figure 2. A client process, using functional addressing sends a request to one instance of a service, then retransmits to another.

a port in Amoeba. The client application's operating system kernel, which runs the communication protocol, always finds out a unique name for a specific process implementing the service before sending the request addressed to that specific process. Amoeba's session identifiers consist of the concatenation of the client and server process' unique names.

We have now seen that there is fundamental uncertainty in case of a crash. It is not possible for a client, in every circumstance, to tell whether or not a service has crashed and, if it has crashed, whether that was before or after doing the requested work. Crashes, however, are rare occurrences and protocol designers should design for the normal case to behave optimally. This means that, in the normal case — that is when there is no crash — a client should get to know that a request has been carried out correctly. That this is possible is illustrated in Figure 3.

In Figure 3-a, we see that a response sent to the client after the complete execution of a request is a positive indication that the request was carried out without failure. But we cannot achieve the reverse at the same time. It is not possible, even with an elaborate protocol, to get a result that is fundamentally better than the one shown in Figure 3-b: when the client does not get a response, it does not know whether the request was executed correctly or perhaps not executed at all.

But we can do a little bit better. It is possible to provide useful feedback to a client in many situations. For instance, it is possible that a client is informed that the network is broken, or that the server has crashed, or that the request could not be delivered. But note that only in the last example — notification of not being
Figure 3. (a.) The correct reception of the response to a request tells the client there were no failures during its execution. (b.) The reverse does not hold. When a request is executed without failure it is still possible that the response does not make it back to the client.

able to deliver a request — a client gets positive information that the request was not executed.

A response from a server to a request sent by a client is the only way in which a client process can know that a request has been carried out correctly by a server. (We are still assuming only amnesia crashes and communication failures. As stated before, if servers retain (part of) their memory, the situation is different.) Acknowledgements of messages received or messages not received can be useful to enhance performance, but they never tell a client what it really needs to know: whether the server carried out the work.

Responses, in a sense, are acknowledgements at the application level. A communication protocol cannot decide to send a response. The server application must do it. Whatever machinery communication protocols invoke to make communication more reliable, there must always be an application-level protocol for the true end-to-end checking that all went correctly. Saltzer, Reed and Clark [1984] called this the end-to-end argument. A good interpretation of the argument is: “Look, we need an application-level end-to-end protocol to make things reliable anyway, so why not integrate this end-to-end machinery with the rest of the protocol machinery for reliable message delivery.”
4 Types of Data Transport

Many request/response protocols have been created that do this exactly and some of the implementations have shown that this can benefit both the usefulness of the protocols as the performance.

To sum up: Communication between processes exhibiting amnesia crashes over networks exhibiting omission failures can be made perfectly reliable except for fundamental uncertainty about the delivery of the last message in a session that is broken off by a failure. Sessions mark a period during which the communicating parties maintain mutual state. Sessions can be terminated gracefully and then there need be no uncertainty about outstanding messages, or it can be terminated abruptly by a crash. Positive feedback about the delivery and processing of messages can only be provided by an end-to-end application-level acknowledgment.

4 Types of Data Transport

Transport protocols in distributed systems are often tailored for a specific class of applications. The diversity of communication requirements prohibits the use of a single protocol that is useful under all circumstances. We can distinguish between four broad classes of communication protocols:

- **Remote Operations**,
- **Bulk Data Transfer**,
- **One-to-Many Communication**, and
- **Continuous Media**.

A remote operation is the most basic form of communication in distributed systems. One process sends a message to another, asking it to do some work. The other process carries out the work and returns a message with the result. The first process is referred to as the *client* process and the second as the *server* process, but note that these terms only refer to the rôles these processes play during the remote operation. One process may simultaneously be a client and a server. Note also that the return message need be nothing more than an acknowledgement that the work has been carried out or merely that the request has been received.¹

Remote operations are often made to look like procedure or function calls. *Remote procedure calls* are, in fact, remote operations with an extra layer of software that packages procedure arguments in request and response messages. Remote procedure call today forms an essential item in the distributed system designer's toolbox. Section 6 is completely devoted to the issues of RPC.

¹. Cf. end-to-end argument in the previous section.
Remote operations are often referred to by other names. Common names are remote invocation, client/server communication, and request/response communication. Sometimes the term RPC is misused to indicate only remote operations.

Remote operations are used for communication between user processes and system services, but also between processes within a single distributed application. The current trend in operating system development towards the use of microkernels and user-level servers has made remote operations an essential ingredient of modern distributed system interfaces.

Bulk data transfer protocols specialize in transferring very large bodies of data efficiently. They are often used as part of file-transfer protocols, such as FTAM, an ISO standard. Bulk data transfer can be viewed as a special form of remote operation, since it is seldom the case that the data does not either come from or go to the client process (the process instigating the transfer). When data goes to the client, the operation can be viewed as a read operation and when data comes from the client process, a write operation.

There is no reason why transport protocols for remote operations should not be designed to be able to carry very large request or response messages, so that bulk data transfer can be done just as efficiently using request/response protocols as bulk-data-transfer protocols. In fact, the end-to-end argument of the previous section argues that having a bulk-data-transfer protocol as a separate protocol function still requires an additional application-level protocol for end-to-end feedback. The combination of the two is just a request/response protocol that happens to be capable of transporting very large messages.

Services that use replication to tolerate failures need large amounts on internal communication to keep the replicas consistent. It is important that this communication reaches all of the replicas in a well-defined order and that agreement exists on which replicas are up and which are down. For such communication an extensive variety of broadcast protocols have been established during the last decade.

Protocol designers tend to distinguish between broadcast and multicast. Broadcast means sending to all hosts on the network; multicast means sending to a selected subset of hosts. In practice, replication protocols are always multicast protocols. The designers of such protocols, however, usually reason from a universe that only contains the members of the multicast group, so for them it makes sense to call them broadcast protocols.

When making a practical implementation of a broadcast protocol, attention must be paid to avoiding unintentionally synchronized messages. In a protocol that makes use of the hardware broadcast facility of an Ethernet, for instance, it would not be a good idea for all the recipients of a broadcast message to return an
acknowledgement immediately; the acks would result in one big collision on the Ethernet that would take some time to sort out. One host, receiving large numbers of packets simultaneously, may lose some due to overload of its network controller.

Several protocols for various forms of broadcast exist. One of the earliest systems containing reliable broadcast protocols with a variety of ordering properties (causal order, total order within a group, global order) was ISIS, a system developed at Cornell University (Birman and Joseph [1987]). In recent years, protocols have also been developed for Amoeba (Tanenbaum, Kaashoek and Bal [1992]) and there appears to be a trend to include multicast protocols in many other systems as well.

Networks for digital telephony have been in existence for a long time now. These networks are monomedia networks, of course, but telephone companies have nonetheless built up a great body of expertise delivering data at a constant rate with very small latencies. All telephony uses virtual circuits which can only be established if the necessary network bandwidth and switch capacity can be guaranteed. Since all circuits require equal bandwidth, this is not unreasonably difficult. The difficulties of managing telephone networks do not stem from the complexity of its data, but from the gigantic size of the network.

Multimedia networks have to combine continuous media transport, which requires low-latency and constant-rate data transport and "normal" data traffic which is very bursty and also requires low-latency delivery. ATM networks appear to be good at allowing the mixture of these data types.

Opinions differ wildly, however, when it comes to guaranteeing the constant latency of the continuous-media traffic. One school of thought maintains that the capacity of the network should be so large that one type of traffic does not interfere with another. Another spends large amounts of thought on prioritizing network traffic and bandwidth-reservation algorithms that guarantee the timely delivery of continuous-media traffic even at high network loads.

The truth, as usual, is probably somewhere in the middle. Taking data networks as an example, we see that early on similar battles were fought over fair allocation of bandwidth to the various customers. Today, local networks do not implement any access restrictions — the network has enough bandwidth to satisfy all traffic during normal operation. Long-haul networks are exploited on a commercial basis, so there is accounting of its use. In high-speed long-haul networks, this by itself may be enough to keep customers from monopolizing the network. In low-speed long-haul networks, such as the ones covering most of Europe, PTTs do have bandwidth-reservation schemes in place, but customers complain bitterly about the service just the same.
5 Transport Protocols for Remote Operations

In the previous sections, some of the desirable properties of transport protocols were discussed. We list them here.

- At-most-once behaviour,
- Positive feedback when no failures occur,
- Error report when there may have been failures,
- Low end-to-end latency, and
- Support for very large request and reply messages.

In this section, we investigate how such protocols can be built and what design aspects contribute to low overhead and high performance.

Achieving at-most-once behaviour requires that the client and server maintain state which allows the client to generate requests in such a way that the server can always tell requests that it has processed already from those that are new. Doing this requires setting up a session between client and server. As discussed before, requests and responses must be labelled with the session identifier and the session identifier must be distinguishable from all previous session identifiers used. Many transport protocols are very careless about establishing new and unique session identifiers and trust that all old packets will have disappeared by the time a new session starts.

Amoeba (Mullender et al. [1990]) uses session identifiers made up of the unique ports of the client and server processes. Unique ports, for purposes of this discussion, are random numbers drawn carefully enough from a large enough set that the probability of session identifiers clashing is small enough to be ignored.

An interesting protocol for session establishment is the $\Delta T$ protocol designed by Fletcher and Watson [1978]. This protocol makes use of the assumption that packets have a maximum lifetime in the network, $\Delta T$, which is composed of the transmission delay through the network, the maximum number of retransmissions of a packet, and the time interval between retransmissions. The protocol dictates that when there is no communication, or no successful communication, for longer than $\Delta T$, the current session automatically terminates and a new one starts. Since packets have a maximum lifetime of $\Delta T$, no packets from a previous session can arrive in the new one.

In the $\Delta T$ protocol, no session identifiers are necessary, obviating the need for random number generators, stable storage, or battery-backed-up clock. There is no

---

1. The time between retransmissions must, of course, be longer than the time it normally takes for an acknowledgment to come back.
connection setup, the protocol just starts numbering packets from zero at the start of a new session. The price paid is that there is a time-out between sessions, but new sessions are only started when there was no communication for a time greater than \( \Delta T \) anyway, or when a host has crashed. After a host comes back up, it enforces the timeout by waiting for a time \( \Delta T \) before accepting or sending messages.

Each remote operation must be part of a session. Request and reply messages must be distinguishable so that it is possible to tell new messages from retransmissions of already accepted ones. There are several ways of doing this.

1. Packets can be numbered separately for each direction, independently of the actual remote operations.
2. Messages can be numbered and a subnumbering scheme can be used for the packets within a message.
3. Remote operations can be numbered and message types and packet subnumbering can be used for distinguishing packets within a remote operation.

The third method is the most practical for high-performance remote-operations transport protocols for reasons explained below.

In Figure 4, a protocol is shown that acknowledges each message. The protocol sends a request (for the time being, we assume messages fit in a single packet) which is acknowledged before the server starts processing it. When the server finishes it sends a reply which is also acknowledged. In order to detect server crashes, the client, after receiving the initial ack, periodically sends a keep-alive message which the server acks.

Most requests are serviced quickly so that no keep-ales are actually sent. In fact, very many requests are serviced so quickly that the response message actually has to wait in the transmission queue for the ack to the request to go out. Similarly, clients often send several requests in a row sufficiently quickly for the next request to have to wait for the ack to the previous reply.

This suggests another protocol, optimized for "fast operations", illustrated in Figure 5. A reply doubles as an ack for the associated request and the client's next request doubles as an ack for the previous reply.

It is obvious how this protocol would work when there are not failures and no messages are lost, and when the supply of requests never ends. But what if these conditions are not fulfilled? If messages are lost, they can be retransmitted and the protocol would still work (request and reply sequence numbers would be used to distinguish originals from duplicates). But the client could never distinguish between a server that just takes extraordinarily long to compute its answer and one that has crashed — no feedback is generated by a server until processing is
Figure 4. A full transport protocol for remote operations; each message is acknowledged and keep-alives are used for detecting server crashes.
Figure 5. A minimal transport protocol for remote operations of short duration; each request is acknowledged by its reply and each reply by the following request.

As an aside, note that numbering remote operations, rather than messages or packets, makes it easy to couple replies to their requests and requests to previous replies. This is why the third message numbering scheme above is the preferred one.

Wouldn't it be nice if we could design a protocol that would work as the protocol of Figure 5 in the normal case and that — in the worst case — would turn into something like the protocol of Figure 4 whenever messages are lost or the server takes a long time to compute a reply? As it turns out, this is quite possible, as shown next.

In the normal case, a request should be acknowledged by its reply, so requests should not be acknowledged immediately. Acks can be postponed using a piggyback timer, a technique that is well known in the networking world. But this means that both the client and the server have to maintain timers — the client needs a retransmission timer and the server a piggyback timer. In networks that lose very few messages, another technique is even simpler.

The client maintains a retransmission timer and retransmits a request when the timer expires. The server does not maintain a timer, but acknowledges any request it receives for the second time. In the most common case — no messages are lost and the server sends a reply within the retransmission period — this obviously works
with a minimum number of messages. In the next common case — no messages are lost and the server takes longer than the retransmission period — the client retransmits the request when its timer expires and the server responds with an ack. The client then knows that the request has arrived and that the server is working on it.

For reply messages the same technique can be used with the roles of client and server reversed: When the client receives a reply for the first time, it does not send an ack. If it has a new request, it sends it to the server and the server interprets that as an ack for the previous reply. If the client has no new request, the server times out and retransmits the reply; the client acknowledges the retransmission.

There is one sticky detail here. Not infrequently, a client invokes a remote operation, receives the reply and exits before the server's retransmission timer expires. The server will then retransmit into the void. Does this matter? Not really. The server cannot distinguish between a client that crashed just before receiving the reply and one that crashed just after receiving the reply anyway. But the time spent retransmitting is, of course, a waste of resources. A simple solution is to let clients, when they terminate, send acknowledgements for all replies that have not been followed up by another request.

One detail still needs to be addressed — detecting server crashes. This can be done as follows. The client state is extended with an "ack-received" flag, which is set to false when a request is transmitted. When the client receives an ack it sets the flag to true and it increases the retransmission timeout period. When the retransmission timer expires, the client checks the flag. If it is false it retransmits the request. If it is true it retransmits only the header of the request, but not the data. When the server receives a request, it checks the header to see if it had received that request already. If this is the case, it ignores the data portion and returns an ack.

The complete protocol is illustrated in Figure 6 for the case of a remote operation of long duration. Timers are indicated by vertical lines with a black arrow head when the timer expires and open arrow head when the timer is stopped because of the arrival of the expected message. Messages are indicated by the usual diagonal arrows, messages containing data by thick ones, header-only messages by thin ones. Three state variables are indicated, seq is the operation sequence number, timer is the retransmission timer value (which can be set to long or short and also affects running timers), and gotack is the ack-received flag.

So far, one-packet requests and replies have been assumed. Now we shall look at the transmission of very large request and reply messages. Not all remote-operation transport protocols implement large messages. The protocol used in
Figure 6. A remote-operations transport protocol that uses just two messages per operation in the normal case, tolerates message loss and detects server crashes.
Topaz (Schroeder and Burrows [1989]) specializes in one-packet messages. In Topaz, large entities are sent by having several parallel threads simultaneously doing multiple remote operations.

Normally, however, transport protocols for remote operations handle large messages. An obvious technique from the connection-oriented-protocol world is to use a sliding-window protocol for large messages. But sliding-window protocols are complicated and impractical for small messages.

A protocol that has become rather popular is the packet-blast or netblit protocol. An early implementation of such a protocol can be found in VMTCP (Nordmark and Cheriton [1989]). The idea is that some fixed number of packets — a packet blast — is sent and acknowledged as one unit. When the sender gets the ack, it can send the next packet blast. If packets are two kilobytes in size, and a blast is 32 packets, then a megabyte message is 16 blasts.

Packet blasts can easily be incorporated in our one-packet-message remote-operations protocol. Let us assume a request is n blasts in size ($B_1, B_2, ..., B_n$; $n \geq 0$). For $i = 1$ to $n$, the client

1. sends $B_i$,
2. starts a retransmission timer, and
3. waits for expiration of the timer or reception of an ack;
4. if the retransmission timer expires, the client goes back to step 1,
5. if an ack is received, the client cancels the timer and iterates.

The last blast, $B_n$, may be a “partial blast”, when the size of the message is not a multiple of the blast size. The total size of the message should be sent as part of the header, so that the server knows when to start work.

All blasts but the last are acknowledged immediately by the server. The last blast is treated just like the request message in the previous protocol: it is acknowledged by the reply, or, if it is received for the second time, by an ack. The same strategy is used by the response message.

Figure 7 shows the protocol for a large request message, a quick response, and a small reply message.

6 Remote Procedure Call

An overwhelming proportion of interactions between processes in a distributed system are remote operations — one process sends a message to another with a request of some kind and the other process returns a reply or an acknowledgement.
Figure 7. The packet-blast protocol integrated in the remote-operations protocol. Shown is a large request message, followed by a quick and short reply.
Interactions of this kind can also be found in centralised systems. System calls can be viewed as operations invoked by a process and carried out by the operating system.

Remote operations have much the same structure as ordinary procedure calls. In making a procedure call, the caller relinquishes control to the called procedure and gets it back when the procedure returns its result. System are, in fact, almost always "encapsulated" in procedure calls. For a Unix programmer, there is little syntactic difference between a call to read and one to strlen — one is a system call, the other is a library routine.

There is much to be said for using the syntax of procedure calls to invoke remote operations. Every programmer is thoroughly familiar with the concept of procedure calls and it nice to have the details of sending and receiving request and reply messages hidden behind the facade of an elegant procedural interface.

Remote procedure call is just that: remote operations in the guise of a procedural interface. In this section we shall discuss how remote procedures work, what the difference is between ordinary procedure calls and remote procedure calls and we shall look at some examples of remote procedure call systems. But in the following section, we shall first show that remote procedure call is not merely syntactic sugar, but that, in many systems, it would be very hard to do without.

6.1 The Problems of Heterogeneity

It is a sad fact of life that different processor types and different programming languages use different representations for the data they manipulate. When a message is sent from one process, written in one programming language and running on one processor type, to another, written in another language and running on another processor type, the contents of the message will no be understood unless there has been prior agreement on the representation of data in the message.

Let us illustrate the problem by an example, shown in Figure 8. In this figure, two data structures are shown containing the same data in the representations of two different processor architectures. The first data item is a string of eight characters, representing the string "Pegasus". The second data item is a 32-bit integer containing the number one million (shown in its hexadecimal representation). The processor on the left has a little-endian architecture: the least-significant byte of an integer has the lowest address — 8, in the example shown. The processor on the right has a big-endian architecture: the least-significant byte of an integer has the highest address — 11, in the example.

When the data structure shown is sent from a little-endian machine to a big-endian one as a byte array, the i'th byte in the little-endian representation will end
up as the n’th byte of the big-endian one. The string “Pegasus” will arrive correctly, but the number one million with hexadecimal representation \(00-0f-42-40\) will arrive as \(40-42-0f-00\) which is quite a different number. If, on the other hand, the data structure is sent as a 32-bit-word array, the number one million will go across correctly, but the string “Pegasus” will end up looking like “ageP sus” which can’t be right.

The point being made here is this: When transmitting data from a machine with one data representation to one with another, one must know the structure of that data. A transport protocol, therefore, can only present data in the representation of the receiving machine if it knows the exact structure of the data.

Most transport protocols execute in the operating system, but carry user-defined data structures, so they cannot be used to do data conversion. Note, that the interpretation of protocol headers does not present a problem, because the protocol knows the exact lay out of the header. It is the data portion of a message that is the problem.

Similar problems occur when process written in different programming languages communicate data structures. Even programs compiled using different compilers for the same programming language can give problems. The size of integers may be different, for instance, or the way in which data items are aligned to byte, word, or long-word boundaries.

### 6.2 RPC Structure

The notion of Remote Procedure Call was introduced by Birrell and Nelson [1984] to deal with the problems of heterogeneity discussed in the previous section and to
provide a pleasant interface to program to.

The components of a remote procedure call mechanism are illustrated in Figure 9. An application which we shall call the client calls a subroutine which is executed in another application, the server. If the client and the server applications were one and the same, the client would call the subroutine directly. Here, however, the client calls another subroutine, called the client stub. This subroutine has exactly the same interface as the server's subroutine, but it is implemented by code that asks the server to execute the subroutine and returns to the client the values it gets back from the server.

The client stub copies the parameters from the stack into a request message. It then calls on a transport protocol to ship the request message to the server. At the server end it is received by another special piece of code, the server stub. The server stub takes the call parameters out of the request message, pushes them onto the stack and calls the actual remote subroutine. When it returns, the whole procedure repeats itself in the reverse direction: The server stub puts the result parameters in a reply message, and invokes the transport protocol to ship the message back to the client stub. The client stub takes the result parameters and passes them back to the client. For the client code the whole thing looks exactly like a normal local procedure call.

Let us go over this mechanism again by looking at an example.

Assume that we want to use an RPC interface between a time server and its clients. It has an interface consisting of two calls as shown in Figure 10.

The client stub, in this case, would consist of two subroutines with the same interface as the remote-procedure one. The one for settime could look like the code of Figure 11.

The client stub shown declares a message buffer and fills it with a code that identifies the call to be made and its parameter: the contents of the struct time. It then sends the request message using the transport protocol's do_operation call. When the transport protocol returns the constant SUCCESS the call succeeded and the reply message will have been put in the buffer message. From this buffer it retrieves a code that indicates whether or not the call succeeded which is returned.

Let us now look at the code for the client stub in more detail. The first thing to note is that the client stub is written using knowledge of the semantics of the settime call: The contents of the struct time pointed to by the parameter t is copied into the request message, but not copied back from the reply message. The implementor of the stub used the knowledge that settime sets the time and does not read the time. Settime returns an integer indicating whether setting the time succeeded. The implementor uses this knowledge and has written code that
Figure 9. Structure of remote procedure call.
6.2 RPC Structure

```c
struct time{
    int seconds;
    int minutes;
    int hours;
    int day;
    int month;
    int year;
    char timezone[4];
}

int gettimeofday(t); struct time *t;
int settimeofday(t); struct time *t;
```

Figure 10. The time server interface uses a data structure called time which is passed in the two calls gettimeofday and settimeofday.

```c
int settimeofday(t); struct time *t; {
    char *p,*message[32];
    int stat;

    p = message;
p = put_int(p, SETTIME);
p = put_int(p, t->seconds);
p = put_int(p, t->minutes);
p = put_int(p, t->hours);
p = put_int(p, t->day);
p = put_int(p, t->month);
p = put_int(p, t->year);
p = put_string(p, t->timezone, 4);
    stat = do_operation("time_server", message, 28);
    if (stat == SUCCESS) get_int(message, &stat);
    return(stat);
}
```

Figure 11. A possible implementation of the client stub for settimeofday.
Remote Procedure Call

retrieves this status code and returns it to the caller.

The second thing to notice is that specific calls to routines called `put_int`, `put_string`, and `get_int` are made to copy integers and strings into and out of messages. These routines are *marshalling routines* and their task is to convert data types from the machine's representation into a standard network representation. We shall see more about marshalling later and network representations in a later section.

Finally, the client stub calls a transport-protocol function `do_operation` that sends a request message from client to server and returns a reply message from the server (in the same buffer, in our example). `do_operation` returns a status code that indicates whether the remote invocation of the server succeeded. In this example, we have overloaded the error indications of the transport mechanism with those of the `settime` call itself. This only works if the implementors of `settime` use a disjoint set of error codes (e.g., zero for success, negative values for transport errors and positive values for error in calls to `settime`).

Now, let us look at a possible implementation of the server stub code, shown in Figure 12.

The server stub has a very different structure from the client stub. The client stub is a subroutine that is called by the client to handle a specific call (`settime` in our example). The server stub is really a *process* that receives calls from all sorts of clients, figures out which subroutine in the RPC interface must be called and calls them, doing the marshalling in the process.

Our example server consists of an infinite loop, each iteration of the loop handling one RPC call. At the top of the loop we find a call to the transport protocol's `receive_request` function. At the bottom we find a call to `send_reply`. In between, the RPC is carried out. First, the operation code is picked out of the message and a *case statement* selects the marshalling code for a specific call in the interface. We only show the marshalling code for `settime` in the example.

After marshalling the arguments of `settime` into a local `struct time t`, the actual `settime` routine is called. `settime` does not modify its arguments, so they need not be marshalled again for the reply message. The return code from `settime` is marshalled, however, and sent back to the client.

Now that we have seen how remote procedure call works, we shall go into several aspects of RPC in more detail. The next section will describe the difference between normal procedure calls and remote ones. Section 6.4 will look at marshalling in more detail. In Section 6.5 the automatic generation of stubs from a description of the RPC interface is explained.
6.2 RPC Structure

```c
void main_loop();
    char *p,*message[32];
    int len, op_code;
    struct time t;

    for (;;) {
        len = receive_request(message, 28);
        if (len < 4) {
            /* error handling code */
        }
        p = message;
        p = get_int(p, op_code);
        switch(op_code) {
            case SETTIME:
                if (len < 32) {
                    /* error handling code */
                }
                p = get_int(p, &t.seconds);
                p = get_int(p, &t.minutes);
                p = get_int(p, &t.hours);
                p = get_int(p, &t.day);
                p = get_int(p, &t.month);
                p = get_int(p, &t.year);
                p = get_string(p, &t.timezone, 4);
                len = settime(&t);
                put_int(message, len);
                len = 4;
                break;
            case GETTIME:
                /* code for marshalling and calling
                * gettime
                */
                send_reply(message, len);
        }
    }
```

**Figure 12.** A possible implementation of the server stub for the time server.
6 Remote Procedure Call

6.3 Procedure Call vs. Remote Procedure Call

The important thing to know about remote procedure call is that it is not the same as ordinary procedure call. There are fundamental differences. They all have to do with the fact that the caller of the procedure and the called procedure execute in different domains.

The most important difference is perhaps that client and server can fail independently. The server may crash while the client does not. As a consequence an RPC may fail. When a crash occurs during the execution of a normal procedure call, the caller crashes also, so no code needs to be written to deal with server crashes. In the remote-procedure-call case, such code will have to be present. If the programming language into which RPC is embedded has an exception-handling mechanism, such as in the case of Modula-2, extra exceptions can be declared for the case of communication or server failures. In other languages, such as, for example, C, the failure codes are usually overloaded on the return values from the remote procedures themselves. Obviously, this restricts the value space of return types.

Client and server do not execute in the same address space, so global variables and pointers cannot be used across the interface. The set of global variables of the client is not accessible from a remote procedure. Pointer passing is useless too. A pointer has meaning only in one address space. In some cases, instead of a pointer, one can pass the data pointed to, as we did in the set time example of the previous section. But this does not always work. If a pointer is passed to an element in the middle of a singly-linked list, should one marshall one element in the linked list, the whole linked list, or just the tail of the list starting at the element pointed to? It depends on what the remote procedure does.

Passing functions as arguments to a procedure is normally implemented by passing a pointer to the function. In an RPC interface, function-passing is close to impossible. One could try to pass the whole function, but this can only work in very favourable circumstances indeed — homogeneous hardware, position-independent code, no dependencies on global variables, and probably quite a few more.

6.4 Marshalling

Marshalling is the technical term for transferring data structures used in remote procedure calls from one address space to another. Marshalling is needed to achieve two goals. One is linearizing the data structures for transport in messages and reconstructing the original data structures at the far end. The other is converting data structures from the data representation on the calling process to that of the called process.
When linearizing data structures, the most complicated problem is to recognize when a structure pointed to has already been marshalled. It is possible that first a pointer to a character is encountered and later a pointer to the string containing that character. It is these complications that make marshalling complex data structures in such a way that they can be reconstructed at the other end quite difficult.

One way of dealing with pointers is not to marshal the data pointed to, but instead to generate a call-back handle. The idea is that the called procedure, when it runs into a pointer, is made to make an RPC back to the caller to retrieve the data pointed to. This approach avoids unnecessary copying of data across the interface and is therefore particularly suitable when passing pointers into very large data structures. However, multiple pointers into the same data will usually go unrecognized with this method.

The marshalling code at the sending and receiving side must use an agreed representation for the data passed in messages between them. This representation is usually identical or nearly identical to the data representation of one of the participating machines (e.g., big-endian, ASCII and IEEE floating point representation) and languages (e.g., integers are four bytes, booleans one bytes, etc.). Sometimes several different representations are allowed to avoid unnecessary translations.

Note that it is unnecessary to use a network standard representation that is self-describing; that is, a representation such as ASN.1 where each character, integer, string, etc. is labelled with its type. The stubs at both ends do not require this information since their code was derived from a common interface definition. In the examples of Figures 11 and 12 this can be observed. `GetInt` and `GetString` are called in the appropriate places without first looking into the message to see what data type comes next.

Variable-length, or variable-structure data types must be self-describing to some extent. Variable-length arrays must be accompanied by the number of elements, unions by a discriminator. Sometimes fixed-length buffers are used for variable-length data. An input routine, such as `getLine`, for example may be called with a buffer of 256 characters which will usually be only partly filled with data. It can be useful to use specialized marshalling code for cases like this and save network bandwidth.

### 6.5 Stub Generation

Stubs as shown in Figures 11 and 12 can simply be hand-crafted from the header declaration of Figure 10 and a little bit of knowledge of what the procedures in the interface do.

It is an attractive notion to generate these stubs purely mechanically from the
7 References

interface description, but descriptions such as that of Figure 10 do not carry sufficient information to do that.

Specialized languages are used to define remote procedure call interfaces and provide the information that existing programming languages do not provide. From an interface definition in such a language, client stubs and server stubs can be generated automatically by a stub compiler.

The language in which a remote procedure call interface is described is usually referred to as an Interface Definition Language or IDL. They tend to be derived from existing programming languages and sometimes, the extra information needed for stub generation for the IDL is made to look like a series of comments in the original programming language.

Essential extra information an IDL should provide is the direction in which parameters travel (in, out, or in/out), discriminators for unions, and information about the length of arrays passed as parameters.

Sometimes, a single server implements multiple interfaces. It is therefore a sensible precaution to make the identifier of the subroutine to be called a global one, for instance by the concatenation of a global interface identifier and an identifier for the function to be called in the interface.

In a combination of a well-designed IDL and a carefully implemented stub compiler, the marshalling code can be very nearly optimal. Examples of IDLs are HP/Apollo’s NCS which is now part of the Distributed Computing Environment of OSF, SUN RPC (Sun Microsystems [1985]), Mercury (Liskov et al. [1987]), Flume (Birrell, Lazowska and Wobber []), Courier (Xerox Corporation [1981]), and AIL (van Rossum [1989]).

7 References


A. D. Birrell, E. D. Lazowska and E. Wobber, Flume—Remote Procedure Call Stub Generator for Modula-2+, Topaz manpage.


G. van Rossum [1989], AIL — A Class-Oriented Stub Generator for Amoeba, Workshop on Progress in Distributed Operating Systems and Distributed Systems Management 433.


