DISTRIBUTED SYSTEMS

Concepts and Design

Fifth Edition

International Edition

George Coulouris
Jean Dollimore
Tim Kindberg
Gordon Blair

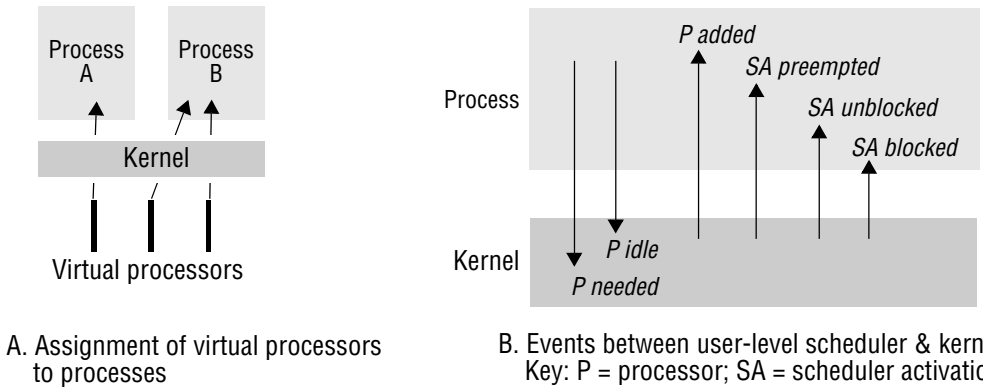
PEARSON

DISTRIBUTED SYSTEMS

Concepts and Design

Fifth Edition

Figure 7.10 Scheduler activations



A. Assignment of virtual processors to processes

B. Events between user-level scheduler & kernel
Key: P = processor; SA = scheduler activation

Figure 7.10(b) shows that a process notifies the kernel when either of two types of event occurs: when a virtual processor is 'idle' and no longer needed, or when an extra virtual processor is required.

Figure 7.10(b) also shows that the kernel notifies the process when any of four types of event occurs. A *scheduler activation* (SA) is a call from the kernel to a process, which notifies the process's scheduler of an event. Entering a body of code from a lower layer (the kernel) in this way is sometimes called an *upcall*. The kernel creates an SA by loading a physical processor's registers with a context that causes it to commence execution of code in the process, at a procedure address designated by the user-level scheduler. An SA is thus also a unit of allocation of a timeslice on a virtual processor. The user-level scheduler has the task of assigning its *READY* threads to the set of SAs currently executing within it. The number of those SAs is at most the number of virtual processors that the kernel has assigned to the process.

The four types of event that the kernel notifies the user-level scheduler (which we shall refer to simply as 'the scheduler') of are as follows:

Virtual processor allocated: The kernel has assigned a new virtual processor to the process, and this is the first timeslice upon it; the scheduler can load the SA with the context of a *READY* thread, which can thus recommence execution.

SA blocked: An SA has blocked in the kernel, and the kernel is using a fresh SA to notify the scheduler; the scheduler sets the state of the corresponding thread to *BLOCKED* and can allocate a *READY* thread to the notifying SA.

SA unblocked: An SA that was blocked in the kernel has become unblocked and is ready to execute at user level again; the scheduler can now return the corresponding thread to the *READY* list. In order to create the notifying SA, the kernel either allocates a new virtual processor to the process or preempts another SA in the same process. In the latter case, it also communicates the preemption event to the scheduler, which can reevaluate its allocation of threads to SAs.

SA preempted: The kernel has taken away the specified SA from the process (although it may do this to allocate a processor to a fresh SA in the same process); the scheduler places the preempted thread in the *READY* list and reevaluates the thread allocation.

This hierarchical scheduling scheme is flexible because the process's user-level scheduler can allocate threads to SAs in accordance with whatever policies can be built on top of the low-level events. The kernel always behaves the same way. It has no influence on the user-level scheduler's behaviour, but it assists the scheduler through its event notifications and by providing the register state of blocked and preempted threads. The scheme is potentially efficient because no user-level thread need stay in the *READY* state if there is a virtual processor on which to run it.

7.5 Communication and invocation

Here we concentrate on communication as part of the implementation of what we have called an *invocation* – a construct, such as a remote method invocation, remote procedure call or event notification, whose purpose is to bring about an operation on a resource in a different address space.

We cover operating system design issues and concepts by asking the following questions about the OS:

- What communication primitives does it supply?
- Which protocols does it support and how open is the communication implementation?
- What steps are taken to make communication as efficient as possible?
- What support is provided for high-latency and disconnected operation?

We focus on the first two questions here then turn to the final two in Sections 7.5.1 and 7.5.2, respectively.

Communication primitives • Some kernels designed for distributed systems have provided communication primitives tailored to the types of invocation that Chapter 5 described. Amoeba [Tanenbaum *et al.* 1990], for example, provides *doOperation*, *getRequest* and *sendReply* as primitives. Amoeba, the V system and Chorus provide group communication primitives. Placing relatively high-level communication functionality in the kernel has the advantage of efficiency. If, for example, middleware provides RMI over UNIX's connected (TCP) sockets, then a client must make two communication system calls (socket *write* and *read*) for each remote invocation. Over Amoeba, it would require only a single call to *doOperation*. The savings in system call overhead are liable to be even greater with group communication.

In practice, middleware, and not the kernel, provides most high-level communication facilities found in systems today, including RPC/RMI, event notification and group communication. Developing such complex software as user-level code is much simpler than developing it for the kernel. Developers typically implement middleware over sockets giving access to Internet standard protocols – often connected

sockets using TCP but sometimes unconnected UDP sockets. The principal reasons for using sockets are portability and interoperability: middleware is required to operate over as many widely used operating systems as possible, and all common operating systems, such as UNIX and the Windows family, provide similar socket APIs giving access to TCP and UDP protocols.

Despite the widespread use of TCP and UDP sockets provided by common kernels, research continues to be carried out into lower-cost communication primitives in experimental kernels. We examine performance issues further in Section 7.5.1.

Protocols and openness • One of the main requirements of the operating system is to provide standard protocols that enable interworking between middleware implementations on different platforms. Several research kernels developed in the 1980s incorporated their own network protocols tuned to RPC interactions – notably Amoeba RPC [van Renesse *et al*. 1989], VMTP [Cheriton 1986] and Sprite RPC [Ousterhout *et al*. 1988]. However, these protocols were not widely used beyond their native research environments. By contrast, the designers of the Mach 3.0 and Chorus kernels (as well as L4 [Härtig *et al.* 1997]) decided to leave the choice of networking protocols entirely open. These kernels provide message passing between local processes only, and leave network protocol processing to a server that runs on top of the kernel.

Given the everyday requirement for access to the Internet, compatibility at the level of TCP and UDP is required of operating systems for all but the smallest of networked devices. And the operating system is still required to enable middleware to take advantage of novel low-level protocols. For example, users want to benefit from wireless technologies such as infrared and radio frequency (RF) transmission, preferably without having to upgrade their applications. This requires that corresponding protocols, such as IrDA for infrared networking and Bluetooth or IEEE 802.11 for RF networking, can be integrated.

Protocols are normally arranged in a *stack* of layers (see Chapter 3). Many operating systems allow new layers to be integrated statically, by including a layer such as IrDA as a permanently installed protocol 'driver'. By contrast, *dynamic protocol composition* is a technique whereby a protocol stack can be composed on the fly to meet the requirements of a particular application, and to utilize whichever physical layers are available given the platform's current connectivity. For example, a web browser running on a notebook computer should be able to take advantage of a wide area wireless link while the user is on the road, and then a faster Ethernet or IEEE 802.11 connection when the user is back in the office.

Another example of dynamic protocol composition is use of a customized request-reply protocol over a wireless networking layer, to reduce round-trip latencies. Standard TCP implementations have been found to work poorly over wireless networking media [Balakrishnan *et al.* 1996], which tend to exhibit higher rates of packet loss than wired media. In principle, a request-response protocol such as HTTP could be engineered to work more efficiently between wirelessly connected nodes by using the wireless transport layer directly, rather than using an intermediate TCP layer.

Support for protocol composition appeared in the design of the UNIX Streams facility [Ritchie 1984], in Horus [van Renesse *et al.* 1995] and in the x-kernel [Hutchinson and Peterson 1991]. A more recent example is the construction of a configurable transport protocol CTP on top of the Cactus system for dynamic protocol composition [Bridges *et al.* 2007].

7.5.1 Invocation performance

Invocation performance is a critical factor in distributed system design. The more designers separate functionality between address spaces, the more remote invocations are required. Clients and servers may make many millions of invocation-related operations in their lifetimes, so small fractions of milliseconds count in invocation costs. Network technologies continue to improve, but invocation times have not decreased in proportion with increases in network bandwidth. This section will explain how software overheads often predominate over network overheads in invocation times – at least, for the case of a LAN or intranet. This is in contrast to a remote invocation over the Internet – for example, fetching a web resource. On the Internet, network latencies are highly variable and relatively high on average; throughput may be relatively low, and server load often predominates over per-request processing costs. For an example of latencies, Bridges *et al.* [2007] report minimal UDP message round-trips taking average times of about 400 milliseconds over the Internet between two computers connected across US geographical regions, as opposed to about 0.1 milliseconds when identical computers were connected over a single Ethernet.

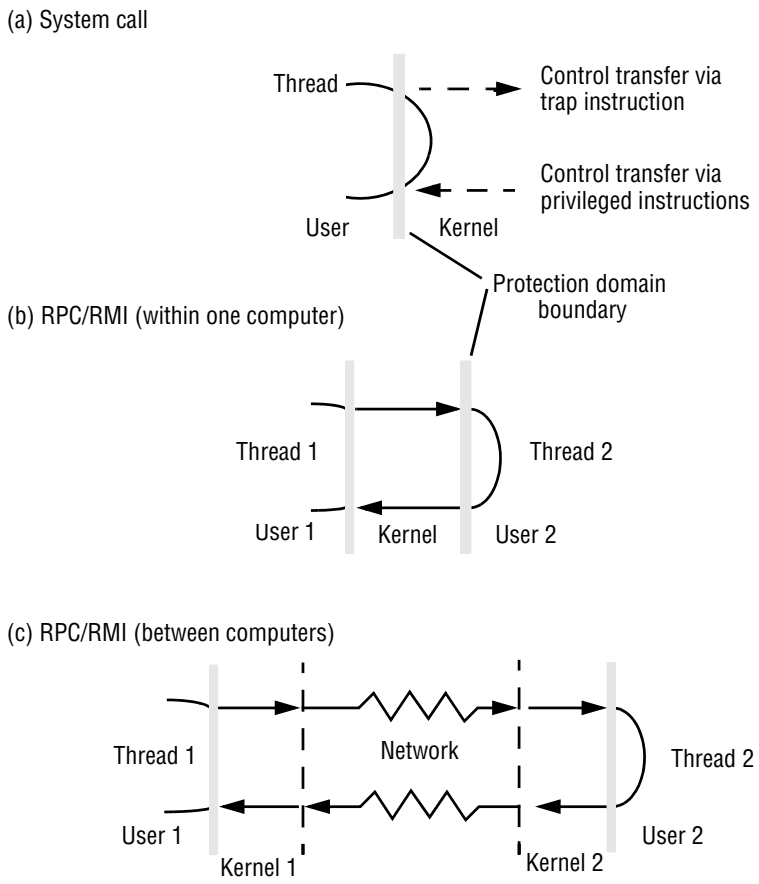
RPC and RMI implementations have been the subject of study because of the widespread acceptance of these mechanisms for general-purpose client-server processing. Much of the research has been carried out into invocations over the network, and particularly into how invocation mechanisms can take advantage of high-performance networks [Hutchinson *et al.* 1989, van Renesse *et al*. 1989, Schroeder and Burrows 1990, Johnson and Zwaenepoel 1993, von Eicken *et al.* 1995, Gokhale and Schmidt 1996]. There is also, as we shall show, an important special case of RPCs between processes hosted at the same computer [Bershad *et al*. 1990, 1991].

Invocation costs • Calling a conventional procedure or invoking a conventional method, making a system call, sending a message, remote procedure calling and remote method invocation are all examples of invocation mechanisms. Each mechanism causes code to be executed outside the scope of the calling procedure or object. Each involves, in general, the communication of arguments to this code and the return of data values to the caller. Invocation mechanisms can be either synchronous, as for example in the case of conventional and remote procedure calls, or asynchronous.

The important performance-related distinctions between invocation mechanisms, apart from whether or not they are synchronous, are whether they involve a domain transition (that is, whether they cross an address space), whether they involve communication across a network and whether they involve thread scheduling and switching. Figure 7.11 shows the particular cases of a system call, a remote invocation between processes hosted at the same computer, and a remote invocation between processes at different nodes in the distributed system.

Invocation over the network • A *null RPC* (and similarly, a *null RMI*) is defined as an RPC without parameters that executes a null procedure and returns no values. Its execution involves an exchange of messages carrying some system data but no user data. The time taken by a null RPC between user processes connected by a LAN is on the order of a tenth of a millisecond (see, for example, measurements by Bridges *et al.* [2007] of round-trip UDP times using two 2.2GHz Pentium 3 Xeon PCs across a 100 megabits/second Ethernet). By comparison, a null conventional procedure call takes a

Figure 7.11 Invocations between address spaces



small fraction of a microsecond. Approximately 100 bytes in total are passed across the network for a null RPC. With a raw bandwidth of 100 megabits/second, the total network transfer time for this amount of data is about 0.01 milliseconds. Clearly, much of the observed *delay* – the total RPC call time experienced by a client – has to be accounted for by the actions of the operating system kernel and user-level RPC runtime code.

Null invocation (RPC, RMI) costs are important because they measure a fixed overhead, the *latency*. Invocation costs increase with the sizes of arguments and results, but in many cases the latency is significant compared with the remainder of the delay.

Consider an RPC that fetches a specified amount of data from a server. It has one integer request argument, specifying how much data to return. It has two reply arguments, an integer specifying success or failure (the client might have given an invalid size) and, when the call is successful, an array of bytes from the server.

Figure 7.12 RPC delay against parameter size

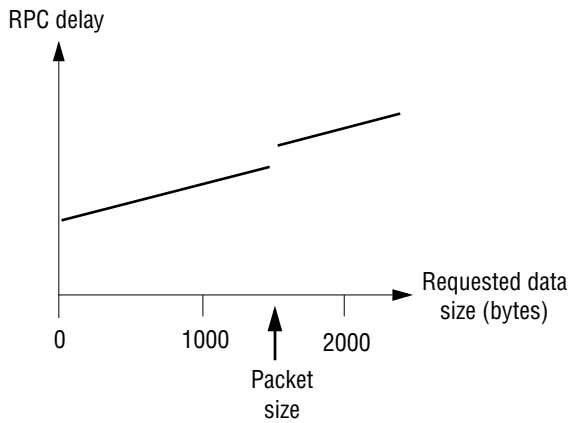


Figure 7.12 shows, schematically, client delay against requested data size. The delay is roughly proportional to the size until the size reaches a threshold at about network packet size. Beyond that threshold, at least one extra packet has to be sent, to carry the extra data. Depending on the protocol, a further packet might be used to acknowledge this extra packet. Jumps in the graph occur each time the number of packets increases.

Delay is not the only figure of interest for an RPC implementation: RPC *throughput* (or bandwidth) is also of concern when data has to be transferred in bulk. This is the rate of data transfer between computers in a single RPC. If we examine Figure 7.12, we can see that the throughput is relatively low for small amounts of data, when the fixed processing overheads predominate. As the amount of data is increased, the throughput rises as those overheads become less significant.

Recall that the steps in an RPC are as follows (RMI involves similar steps):

- A client stub marshals the call arguments into a message, sends the request message and receives and unmarshals the reply.
- At the server, a worker thread receives the incoming request, or an I/O thread receives the request and passes it to a worker thread; in either case, the worker calls the appropriate server stub.
- The server stub unmarshals the request message, calls the designated procedure, and marshals and sends the reply.

The following are the main components accounting for remote invocation delay, besides network transmission times:

Marshalling: Marshalling and unmarshalling, which involve copying and converting data, create a significant overhead as the amount of data grows.