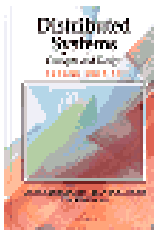


Originally published at pp. 603-8 of Coulouris, Dolllimore and Kindberg, Distributed Systems, Edition 2, 1994.

Firefly RPC

Firefly RPC was designed as a vital component of the new software developed for the Firefly multiprocessor at the DEC Systems Research Centre. It was intended that RPC would be the primary means of communication between processes in the same or different computers and also for system calls. The designers took an approach in which the performance of an RPC was optimized for the 'fast path' – the path taken by the vast majority of RPCs. This implies that minority activities such as dealing with multipackets and retransmissions should not intrude on the fast path, so that the implementation is optimized for the normal case.

Schroeder and Burrows [1990] describe the steps in the fast path, giving times for each of those steps. They provide a model of how time is spent in an average RPC. The emphasis is on the performance of RPCs between different computers. Section 6.5 introduced the main costs involved in an RPC; this section extends that analysis.

Firefly RPC makes use of client and server stub procedures generated from an interface definition written in Modula2+, an extension to Modula2 designed for use in distributed systems. The steps in the Firefly RPC fast path are as follows:

- the client program calls a remote procedure and control passes to a client stub procedure which (1) obtains a packet buffer with a partially filled-in header, (2) marshals the call parameters into a message, (3) sends the request message, and (4) receives and unmarshals the reply;
- at the server, generic RPC run-time code receives the incoming request and looks up and calls the appropriate server stub;
- the server stub (1) unmarshals the request message, but keeps the message buffer, (2) calls the designated procedure, and (3) marshals the reply into the saved message buffer and sends the reply to the client.

Note that although Firefly RPC deals with message retransmissions, acknowledgments, multipacket request and reply messages and server threads, these mechanisms do not belong to the 'fast path' and need not be considered as part of the model for a normal RPC.

Marshalling ◊ Marshalling (and unmarshalling) is normally carried out by Modula2+ stub procedures, which copy arguments to request messages and results from reply messages. Complex types can be marshalled by calling marshalling procedures in a library. Measurements of marshalling overheads were taken for the Firefly RPC implementation, by subtracting the time taken for a null RPC to a separate local address space from the time taken for a similar RPC but with given arguments. The Firefly implementation is such that the difference is accounted for only by the costs of both marshalling and unmarshalling.

The marshalling times for a variable length array passed by reference as a return parameter are given as 115 microseconds for a one-byte array, and 550 microseconds for a 1440-byte array. These figures are for a microVAX II. The increase in marshalling time with size was found to be linear for the case of arrays given, as long as they were shorter than the maximum that would fit in a single packet.

Firefly RPC reduces marshalling costs, by not marshalling data unnecessarily, and by eliminating copying steps. In particular, most parameters are passed in one direction only.

Figure 1 The component costs of Firefly remote procedure calls.

Procedure	*Action*	*Time in microseconds*
Null( )	Stubs and RPC run-time	606
	Send+receive 74-byte call packet	954
	Send+receive 74-byte result packet	954
	TOTAL	2514
MaxResult(*buffer*)	Stubs and RPC run-time	606
	Marshal 1440-byte result packet	550
	Send+receive 74-byte call packet	954
	Send+receive 1514-byte result packet	4414
	TOTAL	6524

Modula2+ interface definitions distinguish between parameters that are passed in both directions, *IN* parameters, which are transmitted from client to server and *OUT* parameters, which are transmitted from server to client. *IN* parameters are not marshalled in the reply message and similarly, *OUT* parameters are not marshalled in the request message.

If an RPC takes place between computers with the same architecture, then data types need not be converted. Firefly monitors when data conversion is not required. If the size of a result parameter is known and is small enough, Firefly reserves space for it in the reply packet, and a pointer to this is passed by the server stub to the server procedure. The procedure can then write the value directly into the message.

Packet initialization ◊ Network packet headers have to be initialized with appropriate values for fields such as the destination address and port number. A multi-layer protocol stack, such as RPC/UDP/IP/Ethernet, requires several headers to be initialized. To some extent, initialization costs can be amortized by using packet headers with pre-initialized values in those fields that remain constant for every packet sent by the process. The Firefly designers suggest that implementing RPC directly over Ethernet packets would save about 100 microseconds per RPC.

In addition to headers, a cost arises from checksum calculations. Each protocol layer may calculate a checksum, which is transmitted with the data for checking at the receiving end. The cost of calculating a checksum increases with the packet size. Figures of 45 microseconds to calculate the checksum for a 74-byte UDP packet, and 440 microseconds for a 1514-byte packet are quoted for the microVAX II.

Shared packet buffers ◊ In the Firefly RPC design, packet buffers are mapped simultaneously and permanently in both the kernel's address space and those of all user processes. Stubs use these buffers as the targets of their marshalling operations, so that no user-to-kernel data copying is necessary. Furthermore, these buffers are accessible by DMA from the network controller, and no copying is required inside the kernel for the data to reach the network controller.

The Firefly RPC buffer pool is shared between all user processes because of the difficulty in answering the following question for any communication design: When an incoming packet arrives, where should the data be placed? The identity of the destination process cannot be ascertained until the packet has been stored and examined. In Firefly RPC, since the buffers are shared between all processes, the kernel can choose any buffer and the problem is avoided. However, this scheme is of limited applicability. Since user processes must be able to allocate and de-allocate buffers from what is a shared pool, this scheme implies trust that no user process will interfere with the buffers or copy private data. The risk is acceptable or non-existent in the case of a single-user workstation or dedicated server computer, but not acceptable for a multi-user computer.

Firefly RPC performance ◊ In order to summarize the relative costs for the actions that go to make up an RPC, we now report the results described by Schroeder and Burrows. These are in fact a mixture of measured and calculated component times, but they are sufficiently accurate when totalled to provide for comparisons between the costs of the components. Two remote procedure calls are considered: *Null*() is a null remote procedure call, and *MaxResult*(*buf*), transfers 1440

Figure 2 The costs, in microseconds, of sending and receiving a packet.

Action	*74-byte packet*	*1514-byte packet*
	– Send–	
Initialize packet headers and checksum	104	499
User-kernel context switch	37	37
Set up message transfer	147	147
DMA transfer to controller	70	815
Ethernet transmission	60	1230
	– Receive –	
DMA transfer to/from controller	80	835
Handle receive interrupt	191	191
calculate UDP checksum	45	440
Wake up RPC thread	220	220
TOTAL	954	4414

bytes of data from the server to the caller's buffer, which is given as the argument *buf*. The Firefly RPCs we shall consider are implemented by an exchange of two packets: the call packet and result packet. The RPCs are between MicroVax-based Firefly multiprocessors over a 10 megabits-per-second Ethernet. We shall not concern ourselves with the multiprocessor nature of the Firefly, and assume a single processor is used at both client and server.

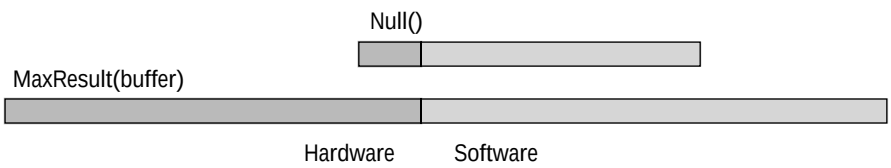
The designers of Firefly RPC went to great pains to optimize their implementation. This included hand-coding critical parts of the code in assembler. The total time for an RPC is made up of three main components:

- the time spent in the client and server stubs to construct the two packets for transmission; this includes buffer allocation and local procedure calling within the RPC run-time software;
- the time to send and receive the call packet; and
- the time to send and receive the result packet.

Figure 1 gives the calculated times, which differ slightly from the measured times. The marshalling of the reply packet is carried out by a stub, but is listed separately for clarity.

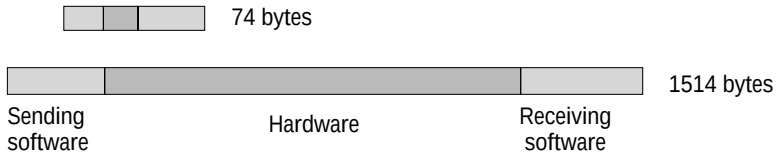
The costs for sending and receiving the packets are as shown in Figure 2. The items in this table are self-explanatory, except that the item 'set up message transfer' involves enqueuing the message for transmission, and communicating with the network controller via a dedicated Firefly processor. Figure 4 summarizes the division between hardware and software costs. Note that the single most costly component of message transmission in the *Null*() case is the time taken to wake up the RPC thread awaiting a packet. The Firefly implementation did not use the optimization of *spinning*, whereby an idle processor remains in the context of the last thread to execute, in case this

Figure 3 Relative hardware and software costs for remote procedure calls.



Note:'Hardware' includes DMA and Ethernet transfer times. 'Software' includes times to communicate with network controllers.

Figure 4 Relative hardware and software costs when sending and receiving a packet.



Note: 'Hardware' includes DMA and Ethernet transfer times. 'Software' includes times to communicate with network controllers.

same thread is the next to be scheduled. This optimization would have reduced this cost considerably.

Considering remote procedure calls overall, we can see that the ratio of software costs to hardware costs varies considerably according to the size of the packets required for the call, and is very high in the case of *Null* (see Figure 3). The hardware costs (DMA and network transmission times taken together) make up about 17 per cent of the overall costs for *Null*, but about 47 per cent of the overall costs for *MaxResult(buffer)*.

The remainder of the time taken by these calls is accounted for by actions taken directly by the processor. Increasing the network bandwidth by a factor of 10, from 10 megabits per second to 100 megabits per second, would reduce the time for *Null* by only about 110 microseconds – just a 4 per cent improvement. On the other hand, increasing the speed of the CPU by a factor of 10 would give a saving for a call to *Null* of about 75 per cent.

As a final point it should be noted that, apart from marshalling, data copying is avoided in the Firefly implementation. As explained above, the shared buffer pool implementation used to achieve this is not applicable to a multi-user environment. Most RPC implementations carry additional memory-to-memory copying overheads.

Summary

The Firefly RPC implementation concentrates upon remote procedure calls between processes residing at different computers. It has been optimized for the common case in which arguments and results are sent in single packets. Data type conversion is omitted except between heterogeneous computers; packet headers are partially pre-initialized; and packet buffers are shared between the kernel and user processes. The resulting performance is good, but not demonstrably better than others (faster implementations exist, but on different processors). Performance would be worse on a multi-user computer, because packet buffers could not be shared. Nonetheless, the implementors took pains to account for the time spent in an RPC. The results confirm the assertion of Chapter 6, that a large proportion of RPC delay is accounted for by the operations of the operating system and RPC run-time, rather than hardware. Scheduling costs, which are a large proportion of the null RPC delay, could have been improved by the technique of spinning.