DISTRIBUTED FILE SYSTEMS

(NOTE : the brevity of this chapter should not be taken as an indication that it isn't important. On the contrary, as distributed computer systems become more common there are many issues which must be reevaluated, and the file system is one which poses particular problems. The chapter is short only because we haven't yet had time to make it longer, and – we hope – better.)

The aim of a computer system is, as we frequently mention, to provide computing services effectively and efficiently. The early systems did so by collecting everything together into a large computer, and letting everyone use all of it. This worked well, provided that the thing you wanted to use was there; it didn't work at all if it wasn't.

Two sorts of thing are in question : devices, and files. If you want to use a specialised piece of machinery – say, a large flat-bed graph plotter – then you must arrange access to one; if you want to use information which has been collected and organised somewhere else, then the same sort of problem arises.

Obviously, there are three solutions. You can go there; you can stay at home, and acquire a similar machine, or a copy of the data; or you can all stay where you are and use the equipment or data by some sort of remote access. It is the last of these possibilities which has grown from occasional connections through telephone wires, to dedicated lines, to computer networks, to high-speed local networks, to distributed computing. (We outlined the path from microcomputers to distributed systems earlier in the chapter *DISTRIBUTED SYSTEMS.*) The ideal is to be able to walk up to any computer and to carry on with work which you might have started on any other computer. This has all sorts of implications for system design; here, we're interested in the file system.

NAMES.

The obvious way to identify files in a distributed system is to use a conventional naming system in each separate computer, and to qualify the file name with the name of the computer if you're using it from somewhere else. For example, if you make a file called `<anysortoffilename>` in Auckland, you would identify it as `<anysortoffilename>` so long as you used it in Auckland, but if you wanted to use it while in Hamilton, you'd call it `<anysortoffilename> at Auckland`. And so on. This is not very hard to do. It is in principle the same as the technique used in Unix to join together two file systems : you *mount* one file system at some point in the directory of the other, when it appears as a part of the extended file system in which all files have one or two common directory names at the beginning of their pathnames. (So we could mount the Auckland file system as a node called `auckland` at the lowest level of the Hamilton system's directory, whereupon our file would be addressed from Hamilton as `/auckland/<anysortoffilename>`.) Actually making it work in practice is a little harder, but still not very hard.

But is that really what we want ? No, it isn't. We want to keep on using the name `<anysortoffilename>` in Hamilton, just as we do in Auckland. This isn't just laziness – there are several good reasons, of which we shall hint at three.

The first is fairly obvious. If we build a file name into a programme, we want the programme to work anywhere in the system if our aim of doing our work anywhere is to be achieved. If we have to change the names, then that won't work. We can imagine automatic schemes which might help with this difficulty – we could have our Auckland compiler always insert `at Auckland` in appropriate places when we compiled programmes, for example, and make sure that the programme would notice if it were executed somewhere else. Unfortunately, if you try to allow for all possibilities, such schemes rapidly become enormously complicated. And once you've made it all work, what happens if you decide to move the file to Hamilton ?

The second follows on from that idea. In fact, it isn't extraordinarily difficult to implement remote operations of the sort we need. (We'll discuss some ways and means in the *HOW TO DO IT* section.) It is much harder to make the system efficient, so that it will always provide access as quickly and cheaply as possible. If you do go to Hamilton and run your programme, then it might well be more economical for the system to move both programme and data file from Auckland to Hamilton and to do all the work there – but this is something that should happen automatically, and perhaps while the programme is running after it has been going long enough for the system to collect the information it needs to work out the economics. It is quite hard to see how to do that sensibly without a location-independent file name.

The third reason is quite different, and raises yet another set of problems. What happens if your system is distributed to the extent that parts of it are disconnected ? This can happen in at least two ways. First, communications lines essential for linking parts of the system can fail. This is, in a sense, a bad reason, because it makes it very difficult to run a location-free system of the sort we want. That's because, in the absence of communications, there is no way to tell whether new files in the separated parts are being given identical names. The only way to guarantee safety is to ban any attempt to make a new filename in the system until communication is restored. This is a pessimistic solution. The optimistic solution is to carry on making new files in the hope that there will be no conflict; people obviously prefer this solution, because their work is not interrupted, but the cost is that the two parts of the file system cannot be joined together as soon as the link is repaired, because a check for duplicates must be carried out and some renaming policy used to resolve any clashes.

The second way to get a disconnected file system is perhaps more interesting, because it is deliberate. As the capacities of portable computers increase, it becomes less and less reasonable to regard them as data collection machines, normally used to gather information which will be copied to a large system for processing. Instead, it is more appropriate to see them as full members of the distributed system. With a system constructed to that view, I can do some work on the distributed system using my portable computer in Auckland, make sure that certain files I will need along the way are moved onto the portable machine's disc, then unplug the machine and trot off to do some work somewhere else. Eventually I arrive at Hamilton, plug my machine in again, and expect to see the rest of the distributed system just as I left it in Auckland. Further, the rest of the distributed system should be able to see my part of the system, at least to the extent that it was visible when I was in Auckland – and it should have been able to say sensible things about the files I took with me while my machine was disconnected.

One solution to these problems is to sit around until the engineers save us by linking everything to everything else with terabyte per second links, thereby guaranteeing instant communication everywhere (for a year or two). That, after all, is more or less how we solved the problems of overloading in single machines. We would feel happier if clever ways to solve the problems using existing facilities could be devised; fortunately, other people think so too, and are engaging in research which they – and we – hope will lead to the methods we seek.

COMPARE :

Silberschatz and Galvin[INT4] : Chapter 17.
