M.I.T. Laboratory for Computer Science
Computer Systems Research Division

April 17, 1980
Request for Comments No. 185

End-to-End arguments in system design

by J. H. Saltzer

Choosing which module should do which function is perhaps the primary activity of the computer system designer, and arguments, rationales, and design principles that provide guidance in this choice of function placement are among the most important tools of a system designer. This paper discusses one class of function placement argument that has been used for many years with neither explicit recognition nor much conviction. However, the emergence of the data communications network as a computer system component has sharpened this line of function placement argument by making more apparent the situations in which and reasons why it applies. It seems appropriate, therefore, to characterize the argument more explicitly, so as to examine its nature and to see how general it really is. The argument appeals to application requirements, and leads in the direction of moving function upward in a layered system, closer to the application that uses the function. We begin by considering the communications network version of the argument.

In a system that includes communications, one usually draws a modular boundary around the communications subsystem and defines a firm interface between it and the rest of the system. When doing so, it quickly becomes apparent that there is a list of functions each of which might be implemented in any of several ways: by the communications subsystem, by its client, as a joint venture, or perhaps redundantly, each doing its own version. In reasoning about this choice the requirements of the application often provide the basis for the argument, which goes as follows:

> *The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)*

We call this line of reasoning against low-level function implementation the "end-to-end argument." The following sections examine the end-to-end argument in detail, first with a case study of a typical example in which it is used—the function in question is reliable data transmission—and then by exhibiting the range of functions to which the same argument can be applied. For the case of the data communication system, this range includes encryption, duplicate message detection, guaranteed message delivery, detecting host crashes, and delivery receipts. In a broader context the argument seems to apply to most functions of a computer operating system, including its file system. Examination of this broader context will be easier of we first consider the more specific data communications context, however.

Careful file transfer

Consider the problem of "careful file transfer." A file of N bytes is stored with the aid of a file system, in the disk storage of computer A. Computer A is linked by a data communications network with computer B, which also has a file system and a disk store. The object is to move the file from computer A's storage to computer B's storage without damage, in the face of knowledge that failures can occur at various points along the way. The application program in this case is the file transfer program, part of which runs at host A and part at host B. In order to discuss the possible threats to the integrity of this transaction, let us assume that the following specific steps are involved:

1. At host A the file transfer program calls upon the file system to read the file from the disk, where it resides on several tracks, and the file system passes it to the file transfer program in fixed-size blocks chosen to be disk-format independent.
2. Also at host A the file transfer program asks the data communication system to transmit the file using some communication protocol that involves splitting the data into packets. The packet size is typically different from the file block size and the disk track size.
3. The data communications network moves the packets from computer A to computer B.
4. At host B a data communication program removes the packets from the data communication protocol and hands the contained data on to a second part of the file transfer application, the part that operates within host B.

5. At host B, the file transfer program asks the file system to write the received data on the disk of host B.

With this model of the steps involved, the following are some of the threats to the transaction that a careful designer might be concerned about:

1. The file, though originally written correctly onto the disk at host A, if read now may contain incorrect data, perhaps because of hardware faults in the disk storage system.
2. The software of the file system, the file transfer program, or the data communication system might make a mistake in buffering and copying the data of the file, either at host A or host B.
3. The hardware processor or its local memory might have a transient error while doing the buffering and copying, either at host A or host B.
4. The communication system might drop or change the bits in a packet, or lose a packet or deliver a packet more than once.
5. Either of the hosts may crash partway through the transaction after performing an unknown amount (perhaps all) of the transaction.

How would a careful file transfer application then cope with this list of threats? One approach might be to reinforce each of the steps along the way using duplicate copies, timeout and retry, carefully located redundancy for error detection, crash recovery, etc. The goal would be to reduce the probability of each of the individual threats to an acceptably small value. (Unfortunately, systematic countering of threat two requires writing correct programs, which task is quite difficult, and not all the programs that must be correct are going to be written as part of the file transfer application.) If we assume further that all these threats are relatively low in probability (low enough that the system allows useful work to be accomplished,) brute force countermeasures such as doing everything three times are likely to appear uneconomical.

The alternate approach might be called an "end-to-end check". Suppose that as an aid to coping with threat number one, stored with each file is a checksum that has sufficient redundancy to reduce the chance of an undetected error in the file to an acceptably negligible value. The application program follows the simple steps above in transferring the file from A to B. Then, as a final additional step, the part of the file transfer application residing in host B reads the transferred

file copy back from its disk storage system into its own memory, recalculates the checksum, and sends this value back to host A, where it is compared with the checksum of the original. Only if the two checksums agree does the file transfer application declare the transaction committed. If the comparison fails, something went wrong, and a retry from the beginning might be attempted.

If failures really are fairly rare, this technique will normally work on the first try; occasionally a second or even third try might be required; one would probably consider two or more failures on the same file transfer attempt as indicating that some part of the system is in need of repair.

Now let us consider the usefulness of a common proposal, namely that the communications system provide, internally, a guarantee of reliable data transmission. It might accomplish this guarantee by providing selective redundancy in the form of packet checksums, sequence number checking, and internal retry mechanisms, for example. With sufficient care, the probability of undetected bit errors can be reduced to any desirable level. The question is whether or not this attempt to be helpful on the part of the communication system is in fact useful to the careful file transfer application.

The answer is clear in examining the list of threats to the complete file transfer—threat number four may have been eliminated, but the careful file transfer application must still take on the responsibility of countering the remaining threats, so it will probably continue to provide its end-to-end checksum on the file. And if it does so, it is equally clear that the extra effort expended in the communication system to provide a guarantee of reliable data transmission is only reducing the frequency of retries by the file transfer application; it has no effect on inevitability or correctness of the outcome, since correct file transmission is assured by the end-to-end checksum whether or not the data transmission system is especially reliable.

Thus the argument: in order to achieve careful file transfer, the application program that performs the transfer must supply a file-transfer-specific, end-to-end reliability guarantee (in this case, a checksum to detect failures and a retry/commit·plan.) For the data communication system to go out of its way to be extraordinarily reliable does not reduce the burden on the application program, it only improves its performance.

Thus the amount of effort to put into reliability measures within the data communication system is seen to be an engineering tradeoff based on performance, rather than a requirement for correctness. Note that performance has several aspects here. If the communication system is too unreliable, the file transfer application performance will suffer because of frequent retries following failures of its end-to-end checksum. If the communication system is beefed up with internal reliability measures, those measures presumably have a performance cost, too, in the form of bandwidth lost to redundant data and delay added by waiting for internal consistency checks to complete before delivering the data. There is little reason to push in this direction very far, when it is considered that *the end-to-end check of the file transfer application must still be implemented no matter how reliable the communication system becomes.* The "proper" tradeoff requires careful thought; for example one might start by designing the communication system to provide just the reliability that comes with little cost and engineering effort, and then evaluate the residual error level to insure that it is consistent with an acceptable retry frequency at the file transfer level. It is probably *not* important to strive for a negligible error rate at any point below the application level.

Other examples of the end-to-end argument

The basic argument—a lower level subsystem that supports a distributed application may be wasting its effort providing a function that must by nature be implemented at the application level anyway—can be applied to a variety of functions in addition to reliable data transmission. Perhaps the oldest and most widely known form of the argument concerns acknowledgement of delivery. A data communication network can easily return an acknowledgement to the sender for every message actually delivered to a recipient. The ARPANET, for example, returns a packet known as "Request For Next Message" (RFNM) whenever it delivers a message. Although this acknowledgement may be useful within the network as a form of congestion control (originally the ARPANET refused to accept another message to the same target until the previous RFNM had returned) it was never found to be very helpful to any application using the ARPANET. The reason is that knowing for sure that the message was *delivered* to the target host is not very important. What the application wants to know is whether or not the target host *acted* on the message; all manner of disaster might have struck after message delivery but before completion of the action requested by the message.

The acknowledgement that is really desired is an end-to-end one, which can be originated only by the target application—"I did it", or "I didn't."

> Another strategy sometimes suggested can be useful if the target host is sophisticated enough to implement "recoverable" messages, that is, when it accepts delivery of a message it also accepts responsibility for guaranteeing that the message is acted upon by the target application. This approach can eliminate the need for an end-to-end acknowledgement in some, but not all applications. An end-to-end acknowledgement is still required for applications in which the action requested of the target host should be done only if similar actions requested of other hosts are successful. This kind of application requires a two-phase commit protocol, which involves end-to-end acknowledgements. Also, if the target application may either fail or refuse to do the requested action, and thus a negative acknowledgement is a possible outcome, an end-to-end acknowledgement may still be a requirement.

Another area in which an end-to-end argument can be applied is that of data encryption. The argument here is that if the data transmission system performs encryption and decryption, it must be trusted to manage securely the required encryption keys, the data will be in the clear and thus vulnerable as it passes into the target node and is fanned out to the target application, and the authenticity of the message must still be checked by the application. If the application performs end-to-end encryption, it obtains its required authentication check, it can handle key management to its satisfaction, and the data is never exposed outside the application.

A more sophisticated argument can be applied to duplicate message suppression. A property of some communication network designs is that a message or a part of a message may be delivered twice, typically as a result of time-out-triggered failure detection and retry mechanisms operating within the network. The network can provide the function of watching for and suppressing any such duplicate messages, or it can simply deliver them. One might expect that an application would find it very troublesome to cope with a network that may deliver the same message twice; indeed it is troublesome. Unfortunately, even if the network suppresses duplicates, the application itself may accidentally originate duplicate requests, in its own failure/retry procedures. These application level duplications look like different messages to the communication system, so it cannot suppress them; suppression must be accomplished by the application itself with knowledge of how to detect its own duplicates. Thus the end-to-end argument again: if the application level has to have a duplicate-suppressing mechanism anyway, that mechanism can also suppress any duplicates generated inside the communication network, so the function can be omitted from that lower level. The same basic reasoning applies to completely omitted messages as well as to duplicated ones.

History, and application to other system areas

The individual examples of end-to-end arguments cited in this paper are not original; they have accumulated over the years. The first example of questionable intermediate delivery acknowledgements noticed by this author was the "wait" message of the M.I.T. Compatible Time-Sharing System, which the system printed on the user's terminal whenever the user entered a command. (The message had some value in the early days of the system, when crashes and communication failures were so frequent that intermediate acknowledgements provided needed reassurance that all was well.)

The end-to-end argument relating to encryption was first publicly discussed by Branstad in a 1974 paper[1]; presumably the military security community held classified discussions before that time. Diffie and Hellman [2] and Kent [3] develop the arguments in more depth, and Needham and Schroeder [4] devised improved protocols for the purpose.

The two-phase-commit data update protocols of Gray [5], Lampson and Sturgis [6] and Reed [7] all use a form of end-to-end argument to justify their existence; they are end-to-end protocols that do not depend for correctness on reliability, FIFO sequencing, or duplicate suppression within the communication system, since all of these problems may also be introduced by other system component failures as well. Reed makes this argument explicitly in the second chapter of his Ph.D. thesis on decentralized atomic actions [8].

Much of the debate in the network protocol community over datagrams, virtual circuits, and connectionless protocols turns out to pit an end-to-end argument against a modularity argument that prizes a reliable, FIFO sequenced, duplicate suppressed stream of data as a system component that is easier to build with. This latter point of view has considerable attraction, but, as suggested above, for some applications it is simply insufficient.

A version of the end-to-end argument in a non-communications application was developed in the 1950's by system analysts whose responsibility included reading and writing files on large numbers of magnetic tape reels. Repeated attempts to define and implement a "reliable tape subsystem" repeatedly foundered, as flaky tape drives, undependable system operators, and system crashes conspired against all narrowly focused reliability measures. Eventually, it became standard

practice for every application to provide its own application-dependent checks and recovery strategy; and to assume that lower-level error detection mechanisms at best reduced the frequency with which the higher-level checks failed. As an example, the Multics file backup system [9], built on a foundation of a magnetic tape subsystem format that provides very powerful error detection and correction features, provides its own error control in the form of record labels and multiple copies of every file.

Lampson, in his arguments supporting the "open operating system" [10], uses a kind of end-to-end argument as a justification. Lampson argues against making *any* function a permanent fixture of lower-level modules; the function may be provided by a lower-level module but it should always be replaceable by an application's special version of the function. The reasoning is that for any function you can think of, at least some applications will find that by necessity they must implement the function themselves in order to correctly meet their own requirements. This line of reasoning leads Lampson to propose an "open" system in which the entire operating system consists of replaceable routines from a library. Such an approach has only recently become feasible in the context of computers dedicated to a single application. It may be the case that the large quantity of fixed supervisor function typical of the large-scale operating systems is only an artifact of economic pressures that demanded multiplexing of expensive hardware and therefore a protected supervisor. Most recent system "kernelization" projects, in fact, have focused at least in part on getting function *out* of low system levels [11, 12]. Though this function movement is inspired by a different kind of correctness argument, it has the side effect of producing an operating system that is more flexible for applications, which is exactly the main thrust of the end-to-end argument.

References

[1] Branstad, D.K., "Security Aspects of Computer Networks," AIAA Paper No. 73-427, AIAA Computer Network Systems Conference, Huntsville, Alabama, April, 1973.

[2] Diffie, W., and Hellman, M.E., "New Directions in Cryptography," *IEEE Trans. on Info. Theory, IT-22*, 6, 1976, pp. 644-654.

[3] Kent, S.T., "Encryption-based Protection Protocols for Interactive User-Computer Communication," S.M. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May, 1976. Also available as M.I.T. Laboratory for Computer Science Technical Report, TR-162, May, 1976.

[4] Needham, R.M., and Schroeder, M.D., "Using Encryption for Authentication in Large Networks of Computers," *CACM 21*, 12, 1978, pp. 993-999.

[5] Gray, J.N., "Notes on database operating systems," in *Operating Systems: An Advanced Course*, Volume 60 of *Lecture Notes in Computer Science*, Springer-Verlag, (1978), pp. 393-481.

[6] Lampson, B., and Sturgis, H., "Crash Recovery in a Distributed Data Storage System," working paper, Xerox PARC, November, 1976 and April, 1979. Submitted to *CACM*.

[7] Reed, D.P., "Implementing Atomic Actions on Decentralized Data," unpublished paper presented at Seventh Symposium on Operating Systems Principles, Asilomar, California, December, 1979.

[8] Reed, D.P., "Naming and Synchronization in a Decentralized Computer System," Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1978. Also available as M.I.T. Laboratory for Computer Science Technical Report, TR-205, September 1978.

[9] Reed, D.P., "Processor Multiplexing in a Layered Operating System," S.M. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June, 1976. Also available as M.I.T. Laboratory for Computer Science Technical Report, TR-164, June, 1976.

[10] Lampson, B.W., and Sproull, R.F., "An Open Operating System for a Single-User Machine," *Proc. Seventh Symposium on Operating Systems Principles*, 1979, pp. 98-105.

[11] Schroeder, M.D., Clark, D.D., and Saltzer, J.H., "The Multics Kernel Design Project," *Proc. Sixth Symposium on Operating Systems Principles, Operating Systems Review 11*, 5, November 1977, pp. 43-56.

[12] Popek, G.J., Kampe, M., Kline, C.S., and Walton, E.J., "UCLA Data Secure Unix," *Proc. 1979 NCC*, AFIPS Press, pp. 355-364.