END-TO-END ARGUMENTS IN SYSTEM DESIGN

J.H. Saltzer, D.P. Reed and D.D. Clark*

M.I.T. Laboratory for Computer Science

This paper presents a design principle that helps guide placement of functions among the modules of a distributed computer system. The principle, called the end-to-end argument, suggests that functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level. Examples discussed in the paper include bit error recovery, security using encryption, duplicate message suppression, recovery from system crashes, and delivery acknowledgement. Low level mechanisms to support these functions are justified only as performance enhancements.

Introduction

Choosing the proper boundaries between functions is perhaps the primary activity of the computer system designer. 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 communication 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. This paper articulates the argument explicitly, so as to examine its nature and to see how general it really is. The argument appeals to application requirements, and provides a rationale for moving function upward in a layered system, closer to the application that uses the function. We begin by considering the communication network version of the argument.

In a system that includes communications, one usually draws a modular boundary around the communication subsystem and defines a firm interface between it and the rest of the system. When doing so, it becomes apparent that there is a list of functions each of which might be implemented in any of several ways: by the communication subsystem, by its client, as a joint

---

* Authors' addresses: J.H. Saltzer and D.D. Clark, M.I.T. Laboratory for Computer Science, 545 Technology Square, Cambridge, Massachusetts 02139.: D.P. Reed, Software Arts, Inc., 27 Mica Lane, Wellesley, Massachusetts 02181.

This research was supported in part by the Advanced Research Projects Agency of the U.S. Department of Defense and monitored by the Office of Naval Research under contract number N00014-75-C-0661.

Revised version of a paper from the Second International Conference on Distributed Computing Systems, Paris, France, April 8-10, 1981, pp. 509-512.: Copyright 1981 by The Institute of Electrical and Electronics Engineers, Inc. Reprinted with permission.

Published in ACM Transactions in Computer Systems 2, 4, November, 1984, pages 277-288.

Reprinted in Craig Partridge, editor Innovations in internetworking. Artech House, Norwood, MA, 1988, pages 195-206. ISBN 0-89006-337-0. Also scheduled to be reprinted in Amit Bhargava, editor. Integrated broadband networks. Artech House, Boston, 1991. ISBN 0-89006-483-0.

Scribe/FinalWord source: http://web.mit.edu/Saltzer/www/publications/

venture, or perhaps redundantly, each doing its own version. In reasoning about this choice, the requirements of the application provide the basis for a class of arguments, which go 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, message sequencing, guaranteed message delivery, detecting host crashes, and delivery receipts. In a broader context the argument seems to apply to many other functions of a computer operating system, including its file system. Examination of this broader context will be easier if we first consider the more specific data communication context, however.

End-to-end caretaking

Consider the problem of "careful file transfer." A file is stored by a file system, in the disk storage of computer A. Computer A is linked by a data communication 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 file's integrity in 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 communication 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 part way 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 written by the file transfer application programmer. 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 appear uneconomical.

The alternate approach might be called "end-to-end check and retry". 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 communication 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 useful to the careful file transfer application.

The answer is that threat number four may have been eliminated, but the careful file transfer application must still counter the remaining threats, so it should still provide its own retries based on an end-to-end checksum of the file. And if it does so, 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 and retry 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 to ensure reliability.

A too-real example

An interesting example of the pitfalls that one can encounter turned up recently at M.I.T.: One network system involving several local networks connected by gateways used a packet checksum on each hop from one gateway to the next, on the assumption that the primary threat to correct communication was corruption of bits during transmission. Application programmers, aware of this checksum, assumed that the network was providing reliable transmission, without realizing that the transmitted data was unprotected while stored in each gateway. One gateway computer developed a transient error in which while copying data from an input to an output buffer a byte pair was interchanged, with a frequency of about one such interchange in every million bytes passed. Over a period of time many of the source files of an operating system were repeatedly transferred through the defective gateway. Some of these source files were corrupted by byte exchanges, and their owners were forced to the ultimate end-to-end error check: manual comparison with and correction from old listings.

Performance aspects

It would be too simplistic to conclude that the lower levels should play no part in obtaining reliability, however. Consider a network that is somewhat unreliable, dropping one message of each hundred messages sent. The simple strategy outlined above, transmitting the file and then checking to see that the file arrived correctly, would perform more poorly as the length of the file increases. The probability that all packets of a file arrive correctly decreases exponentially with the file length, and thus the expected time to transmit the file grows exponentially with file length. Clearly, some effort at the lower levels to improve network reliability can have a significant effect on application performance. But the key idea here is that the lower levels need not provide "perfect" reliability.

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 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.

Using performance to justify placing functions in a low-level subsystem must be done carefully. Sometimes, by examining the problem thoroughly, the same or better performance enhancement can be achieved at the high level. Performing a function at a low level may be more efficient, if the function can be performed with a minimum perturbation of the machinery already included in the low-level subsystem, but just the opposite situation can occur – that is, performing the function at the lower level may cost more – for two reasons. First, since the lower level subsystem is common to many applications, those applications that do not need the function will pay for it anyway. Second, the low-level subsystem may not have as much information as the higher levels, so it cannot do the job as efficiently.

Frequently, the performance tradeoff is quite complex. Consider again the careful file transfer on an unreliable network. The usual technique for increasing packet reliability is some sort of per-packet error check with a retry protocol. This mechanism can be implemented either in the communication subsystem or in the careful file transfer application. For example, the receiver in the careful file transfer can periodically compute the checksum of the portion of the file thus far received and transmit this back to the sender. The sender can then restart by retransmitting any portion that arrived in error.

The end-to-end argument does not tell us where to put the early checks, since either layer can do this performance-enhancement job. Placing the early retry protocol in the file transfer application simplifies the communication system, but may increase overall cost, since the communication system is shared by other applications and each application must now provide its own reliability enhancement. Placing the early retry protocol in the communication system may be more efficient, since it may be performed inside the network on a hop-by-hop basis, reducing the delay involved in correcting a failure. At the same time, there may be some application that finds the cost of the enhancement is not worth the result but it now has no choice in the matter* . A great deal of information about system implementation is needed to make this choice intelligently.

Other examples of the end-to-end argument

Delivery guarantees

The basic argument that 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 delivered to a recipient. The ARPANET, for example, returns a packet known as "Request For Next Message" (RFNM)[1] 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 applications 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 for obtaining immediate acknowledgements is to make the target host sophisticated enough that 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[5,10,15], which is a sophisticated end-to-end acknowledgement. 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.

Secure transmission of data

Another area in which an end-to-end argument can be applied is that of data encryption. The argument here is threefold. First, if the data transmission system performs encryption and decryption, it must be trusted to manage securely the required encryption keys. Second, 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. Third, 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

---

* For example, real time transmission of speech has tighter constraints on message delay than on bit-error rate. Most retry schemes significantly increase the variability of delay.

can handle key management to its satisfaction, and the data is never exposed outside the application.

Thus, to satisfy the requirements of the application, there is no need for the communication subsystem to provide for automatic encryption of all traffic. Automatic encryption of all traffic by the communication subsystem may be called for, however, to ensure something else – that a misbehaving user or application program does not deliberately transmit information that should not be exposed. The automatic encryption of all data as it is put into the network is one more firewall the system designer can use to ensure that information does not escape outside the system. Note however, that this is a different requirement from authenticating access rights of a system user to specific parts of the data. This network-level encryption can be quite unsophisticated – the same key can be used by all hosts, with frequent changes of the key. No per-user keys complicate the key management problem. The use of encryption for application-level authentication and protection is complementary. Neither mechanism can satisfy both requirements completely.

Duplicate message suppression

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.

A common example of duplicate suppression that must be handled at a high level is when a remote system user, puzzled by lack of response, initiates a new login to a time-sharing system. For another example, most communication applications involve a provision for coping with a system crash at one end of a multi-site transaction: reestablish the transaction when the crashed system comes up again. Unfortunately, reliable detection of a system crash is problematical: the problem may just be a lost or long-delayed acknowledgement. If so, the retried request is now a duplicate, which only the application can discover. 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.

Guaranteeing FIFO message delivery

Ensuring that messages arrive at the receiver in the same order they are sent is another function usually assigned to the communication subsystem. The mechanism usually used to achieve such first-in, first-out (FIFO) behavior guarantees FIFO ordering among messages sent on the same virtual circuit. Messages sent along independent virtual circuits, or through intermediate processes outside the communication subsystem may arrive in an order different from the order sent. A distributed application in which one node can originate requests that initiate actions at several sites cannot take advantage of the FIFO ordering property to guarantee that the actions requested occur in the correct order. Instead, an independent mechanism at a higher level than the communication subsystem must control the ordering of actions.

Transaction management

We have now applied the end-to-end argument in the construction of the SWALLOW distributed data storage system[15], where it leads to significant reduction in overhead. SWALLOW provides data storage servers called repositories that can be used remotely to store and retrieve data. Accessing data at a repository is done by sending it a message specifying the object to be accessed, the version, and type of access (read/write), plus a value to be written if the access is a write. The underlying message communication system does not suppress duplicate messages, since a) the object identifier plus the version information suffices to detect duplicate writes, and b) the effect of a duplicate read request message is only to generate a duplicate response, which is easily discarded by the originator. Consequently, the low-level message communication protocol is significantly simplified.

The underlying message communication system does not provide delivery acknowledgement either. The acknowledgement that the originator of a write request needs is that the data was stored safely. This acknowledgement can be provided only by high levels of the SWALLOW system. For read requests, a delivery acknowledgement is redundant, since the response containing the value read is sufficient acknowledgement. By eliminating delivery acknowledgements, the number of messages transmitted is halved. This message reduction can have a significant effect on both host load and network load, improving performance. This same line of reasoning has also been used in development of an experimental protocol for remote access to disk records[6]. The resulting reduction in path length in lower-level protocols was important in maintaining good performance on remote disk access.

Identifying the ends

Using the end-to-end argument sometimes requires subtlety of analyis of application requirements. For example, consider a computer communication network that carries some packet voice connections, conversations between digital telephone instruments. For those connections that carry voice packets, an unusually strong version of the end-to-end argument applies: if low levels of the communication system try to accomplish bit-perfect communication, they will probably introduce uncontrolled delays in packet delivery, for example, by requesting retransmission of damaged packets and holding up delivery of later packets until earlier ones have been correctly retransmitted. Such delays are disruptive to the voice application, which needs to feed data at a constant rate to the listener. It is better to accept slightly damaged packets as they are, or even to replace them with silence, a duplicate of the previous packet, or a noise burst. The natural redundancy of voice, together with the high-level error correction procedure in which one participant says "excuse me, someone dropped a glass. Would you please say that again?" will handle such dropouts, if they are relatively infrequent.

However, this strong version of the end-to-end argument is a property of the specific application – two people in real-time conversation – rather than a property, say, of speech in general. If one considers instead a speech message system, in which the voice packets are stored in a file for later listening by the recipient, the arguments suddenly change their nature. Short delays in delivery of packets to the storage medium are not particularly disruptive so there is no longer any objection to low-level reliability measures that might introduce delay in order to achieve reliability. More important, it is actually helpful to this application to get as much accuracy as possible in the recorded message, since the recipient, at the time of listening to the recording, is not going to be able to ask the sender to repeat a sentence. On the other hand, with a storage system acting as the receiving end of the voice communication, an end-to-end argument does apply to packet ordering and duplicate suppression. Thus the end-to-end argument is not an absolute rule, but rather a guideline that helps in application and protocol design analysis; one must use some care to identify the end points to which the argument should be applied.

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 the authors 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[3]. (The message had some value in the early days of the system, when crashes and communication failures were so frequent that intermediate acknowledgements provided some needed reassurance that all was well.)

The end-to-end argument relating to encryption was first publicly discussed by Branstad in a 1973 paper[2]; presumably the military security community held classified discussions before that time. Diffie and Hellman[4] and Kent[8] develop the arguments in more depth, and Needham and Schroeder[11] devised improved protocols for the purpose.

The two-phase-commit data update protocols of Gray[5], Lampson and Sturgis[10] and Reed[13] 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[14].

End-to-end arguments are often applied to error control and correctness in application systems. For example, a banking system usually provides high-level auditing procedures as a matter of policy and legal requirement. Those high-level auditing procedures will uncover not only high-level mistakes such as performing a withdrawal against the wrong account, it will also detect low-level mistakes such as coordination errors in the underlying data management system. Therefore a costly algorithm that absolutely eliminates such coordination errors may be arguably less appropriate than a less costly algorithm that just makes such errors very rare. In airline reservation systems, an agent can be relied upon to keep trying, through system crashes and delays, until a reservation is either confirmed or refused. Lower level recovery procedures to guarantee that an unconfirmed request for a reservation will survive a system crash are thus not vital. In telephone exchanges, a failure that could cause a single call to be lost is considered not worth providing explicit recovery for, since the caller will probably replace the call if it matters[7]: All of these design approaches are examples of the end-to-end argument being applied to automatic recovery.

Much of the debate in the network protocol community over datagrams, virtual circuits, and connectionless protocols is a debate about end-to-end arguments. A modularity argument prizes a reliable, FIFO sequenced, duplicate-suppressed stream of data as a system component that is easy to build on, and that argument favors virtual circuits. The end-to-end argument claims that centrally-provided versions of each of those functions will be incomplete for some applications, and those applications will find it easier to build their own version of the functions starting with datagrams.

A version of the end-to-end argument in a non-communication 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[17], even though it is 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.

The arguments that are used in support of reduced instruction set computer (RISC) architecture are similar to end-to-end arguments. The RISC argument is that the client of the architecture will get better performance by implementing exactly the instructions needed from primitive tools; any attempt by the computer designer to anticipate the client's requirements for an esoteric feature will probably miss the target slightly and the client will end up reimplementing that feature anyway. (We are indebted to M. Satyanarayanan for pointing out this example.)

Lampson, in his arguments supporting the "open operating system,"[9] uses an argument similar to the 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 meet correctly 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 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[16,12]. 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.

Conclusions

End-to-end arguments are a kind of "Occam's razor" when it comes to choosing the functions to be provided in a communication subsystem. Because the communication subsystem is frequently specified before applications that use the subsystem are known, the designer may be tempted to "help" the users by taking on more function than necessary. Awareness of end-to-end arguments can help to reduce such temptations.

It is fashionable these days to talk about "layered" communication protocols, but without clearly defined criteria for assigning functions to layers. Such layerings are desirable to enhance modularity. End-to-end arguments may be viewed as part of a set of rational principles for organizing such layered systems. We hope that our discussion will help to add substance to arguments about the "proper" layering.

Acknowledgements

Many people have read and commented on an earlier draft of this paper, including David Cheriton, F.B. Schneider, and Liba Svobodova. The subject was also discussed at the ACM Workshop in Fundamentals of Distributed Computing, in Fallbrook, California during December 1980. Those comments and discussions were quite helpful in clarifying the arguments.

References

1. Bolt Beranek and Newman Inc. Specifications for the interconnection of a host and an IMP. Technical Report No. 1822, Cambridge, Mass., December, 1981.
2. Branstad, D.K. Security aspects of computer networks. AIAA Paper No. 73-427, AIAA Computer Network Systems Conference, Huntsville, Alabama, April, 1973.
3. Corbato, F.J., et al. *The Compatible Time-Sharing System, A Programmer's Guide*. M.I.T. Press, Cambridge, Massachusetts, 1963, p.10.
4. Diffie, W., and Hellman, M.E. New directions in cryptography. *IEEE Trans. on Info. Theory*, IT-22, 6, (November, 1976), pp.644-654.
5. Gray, J.N. Notes on database operating systems. In *Operating System: An Advanced Course*. Volume 60 of *Lecture Notes in Computer Science*, Springer-Verlag, 1978, pp.393-481.
6. Greenwald, M. Remote virtual disk protocol specifications. M.I.T. Laboratory for Computer Science Technical Memorandum, in preparation. Expected publication, 1984.
7. Keister, W., Ketchledge, R.W., and Vaughan, H.E.: No. 1 ESS: System organization and objectives. *Bell System Technical Journal 53*, 5 (part 1), (September, 1964) p. 1841.
8. 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.
9. Lampson, B.W., and Sproull, R.F. An open operating system for a single-user machine. *Proc. Seventh Symposium on Operating Systems Principles, Operating Systems Review 13*, Special issue (December, 1979), pp.98-105.
10. 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*.
11. Needham, R.M., and Schroeder, M.D.: Using encryption for authentication in large networks of computers. *CACM 21*, 12, (December, 1978), pp.993-999.
12. Popek, G.J., et al.: UCLA data secure unix. *Proc. 1979 NCC*, AFIPS Press, pp.355-364.
13. Reed, D.P.: Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems 1*, 1 (February, 1983), pp.3-23.
14. 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.
15. Reed, D.P., and Svobodova, L.: SWALLOW: A distributed data storage system for a local network. In West, A., and Janson, P., ed. *Local Networks for Computer Communications, Proc. IFIP Working Group 6.4 International Workshop on Local Networks*. North-Holland, Amsterdam, 1981, pp.355-373.
16. 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.
17. Stern, J.A.: Backup and recovery of on-line information in a computer utility. S.M. thesis, M.I.T. Department of Electrical Engineering and Computer Science, August 1973. Available as M.I.T. Project MAC Technical Report TR-116, January, 1974.