M.I.T. LABORATORY FOR COMPUTER SCIENCE April 13, 1979

Computer Systems Research Division Request for Comments No. 170

RELIABILITY ISSUES IN DISTRIBUTED INFORMATION PROCESSING SYSTEMS

by Liba Svobodova

This paper was accepted for presentation at the FTCS-9 (IEEE Fault Tolerant Computing Symposium) to be held in Madison, Wisconsin in June 1979.

The goal of this research is to investigate the matter of reliability in a class of distributed information processing systems composed of highly autonomous nodes connected by a communication network with unpredictable delays. Two classes of issues are discussed: error detection and reporting in internode communication and replication of resources to enhance availability. Attention is paid to the problem of performance degradation that results directly from the application of the mechanisms needed to ensure reliable operation.

1. Introduction

Reliability is and will be one of the major issues in information processing systems. This claim is based on two observations. First, the quantity of information entrusted to a computer system is ever increasing. Second, the complexity of the operations performed by a computer is also increasing. More and more organizations and systems are dependent on computer maintained information and a failure of these computer systems can often be critical. Thus high reliability is not just a requirement for real-time systems controlling space vehicles or industrial process failures of which would endanger human lives.

Reliability of an information processing system is not merely a question of the hardware components. Software errors, synchronization failures, and errors of the human users must be anticipated and handled gracefully. The only way to design a reliable system is to make it "fault-tolerant", or, robust in face of a large variety of internal failures and misuse.

Distributed systems are often claimed to be inherently more reliable than systems based on a large central processor. That is, given that a distributed system is properly designed, it offers better reliability. This claim is

based on several factors. First, distributed systems by their very nature provide opportunities for redundancy. Second, error propagation is restricted by physical separation of processes and resources. And finally, individual nodes in the distributed system may be less complex than a large central processor and, as a result, ought to have lower probability of failures. Basically, distributed systems have a potential for being more reliable than systems based on a large central processor. However, this potential needs to be exploited through proper design.

This paper studies the effect of reliability requirements on the design of distributed information processing systems. In the discussion that follows, the term "reliability" has a very informal meaning; it can be described as the degree to which the system satisfies the expectation that at any point in time the services and resources supported by the system perform their function correctly and are available to the user.* It is difficult to define quantitative measures of reliability, especially when the reliability concerns encompass the integrity and completeness of stored information. Thus, no attempt is made to specify a measure of reliability; rather, reliability is treated here as a complex problem that penetrates all aspects and levels of a system design.

2. Distributed Information Processing Systems

In the areas of real time control, distributed systems have long been viewed as an architecture that enhances reliability. The fault-tolerant

---

* Randell [17] defines reliability as a measure of success with which the system conforms to the specification of its behavior. However, the behavior of a system includes the system's failures! The system specification ought to include the specifications of the situations when the system fails to provide the defined service. Thus, the definition of reliability ought to be restated to say that it is a measure of success with which the system performs its intended (normal, useful) function, as described in the specification.

systems have been built of processing modules with well defined interfaces, where either the individual modules would be designed to perform a specific function [2] or could be engaged to execute any program (or a set of programs) as the need arises. In the latter category, the assignment of tasks to modules could be fixed for a specific configuration and modified when the system has to be reconfigured as a consequence of a failure of one of the modules or connecting buses [4, 25] or dynamic, where the modules would be assigned tasks from a central queue, that is, the modules could be treated as a general pool of processing resources [3, 15]. This paper, however, deals with a significantly different class of distributed systems, systems where individual processing modules can be computers that have not only their private memory, but also their own secondary storage devices and input and output devices. The relative location of these computer systems, called here nodes, can vary from being collocated in the same room to being several thousands of miles apart. The nodes are connected by a communication network that may range from a single multiplexed bus in a local area network [5] to a complex store-and-forward long haul network such as the ARPANET [19]. Most importantly, the individual nodes are highly autonomous, to the point that they may, at the discretion of their owners, discontinue services, restrict access to their local resources, or even completely disconnect themselves from the network to do local maintenance or process some sensitive work. These events may happen when the other nodes in a distributed system least expect them; in fact, they are very similar to a failure; the requesting node will have to be able to deal with the possibly harmful effects arising from the nodes' autonomy in much the same way as it has to deal with the physical failures in the network.

The applications running on such distributed systems can be pictured as a network of intelligent entities, called here agents, communicating by sending messages that contain commands, data, and responses meaningful at this level (Figure 1).* An agent is a program that performs a certain task or tasks that may require cooperation of other agents. Resources, both those defined by the application and those provided by the system, will be modeled as being represented by agents. Resources include hardware devices, software tools, and information. The agents representing resources can be viewed as resource guardians.** Only the guardian has a direct access to the resource it guards. Other agents may use various resources only by sending messages to appropriate guardians requesting specific operations to be performed. The result is an abstract network, or, rather, several levels of such a network.*** Figure 2 shows two levels of the abstract network and the underlying physical network; the two levels are the application defined agents and the communication agents that control the physical delivery of messages among the nodes.

---

* This model has been developed by the Distributed Systems Group at the M.I.T. Loratory for Computer Science. This group has been working on a design of a programming system for development, maintenance and control of distributed applications. The programming system will combine language and facilities traditionally considered to belong to the operating system [21]. The programming language will provide abstraction mechanisms, specifically, it will support data abstractions as it has been done in some experimental languages (CLU [12], Alphard [25]).

** A guardian is similar to a monitor [10]; however, in addition to providing synchronization, a guardian agent can be used to provide protection against unauthorized use and supervise recovery of the resource from faults and misuse.

*** It is assumed that individual resources do not move dynamically from node to node, depending on the degree of demand. Rather, when a specific node is chosen to be the (new) home of a particular resource, an installation of the resource has to be explicitely requested using commands provided by the programming system. This assumption is based on the belief that such placement decisions will often be based on non-technical factors external to the system [21]. This assumption also simplifies the reliability mechanisms that need to be provided in a distributed system.

The discussion of mechanisms pertaining to reliability refers to two levels, the application level and the system level. System level is all the mechanisms needed to support the view presented to the application programmer (that is, the hardware and software run-time support). The level built on the top of this level using the tools available to the application programmer is referred to as the application level. To achieve reliable operations from the application point of view, both the system and the application software have to be properly designed. For each type of error, it is necessary to decide where it can be detected and how it should be handled. Some errors are application dependent and therefore, their detection and handling has to be delegated to the application level. Some classes of errors, detected within the system level can be masked, but others have to be forwarded to the application level. Basically, in the class of system errors, there is a gray area where a decision has to be made as to whether the system will mask the errors, or whether the errors will be delegated to the application level. It may also be possible that an attempt to mask an error fails; thus a decision has to be made as to how long the system should keep trying to correct the error, before passing a notification to the application level. The system must provide means for detecting and correcting or reporting errors arising from the operation of the hardware and the software that supports the application programs. However, the system also has to provide suitable primitives for the application programmer to facilitate handling of the application specific errors and communication of the system detected errors to the application programs. It should be pointed out, however, that the more complex reliability mechanisms are built into a system to support the kind of

or the level of reliability required, the higher the probability that these mechanisms, and consequently the entire system, will fail. Therefore, simplicity is considered to be a virtue.

3. Availability and Correctness

Reliability has two aspects that, unfortunately, cannot always be separated. One aspect is the availability of the resources (including programs and data) needed to perform a specific task. The other is the correctness of the available resources; a very important special case is the integrity of the stored information. Availability has several connotations. First, it is used to indicate whether a resource is useable, that is, if the respective agent will execute the operation requested once the request is brought to its attention (e.g. gets to the head of the queue). Second, availability may also be used to indicate whether a resource can be used immediately, or whether there is a contention for the resource (the resource may be used by somebody else and there may be a queue of requests that precede the particular request of interest). Third, and this aspect plays an important role in a distributed system, it may indicate whether a resource is accessible. A resource may be useable and unused, but the path to it may be broken. However, it is possible to translate all three aspects into the problem of how long it is necessary to wait for a resource. A useable resource may not be immediately available due to contention for the resource, but also due to long communication delays; if the communication path is broken, the communication delays may be unusually long, even infinite. Similarly, if a resource is unuseable, the wait time for the resource to become useable may be very long, possibly infinite. Since in a distributed system it is not always possible to determine the cause of a long delay, the

system may have to respond to poor performance (due to overload) in the same or similar way it responds to functional failures of the resources and communication paths. Thus, in a way, poor performance turns into a failure!

To assure correct operation, the system and the application have to be prepared to handle errors that originate in the lower levels, in particular, hardware faults and possible residual bugs in the software that composes both the system level and the application level. However, the system also has to be prepared to deal with errors where the source is the user of a resource. Since the user may be an agent running on another node, these external errors may be caused by hardware failures or software failures in the user's node, but they also may be caused by the fact that the requesting agent either does not know how to use the requested resource properly or is trying to misuse it intentionally. Thus, to ensure correct operation of a resource, it is necessary to ensure both that the operations on that resource are performed correctly in face of possible failures of the node on which the resource resides, and also, it is necessary to defend it from possible misuse from other agents.

As said earlier, the agents can communicate only by sending messages. A resource can be manipulated only by a single agent designed to be the guardian of the resource. To protect a resource from misuse, it is necessary to ensure that indeed no other agents can gain an access to the resource and that all incoming messages are carefully scrutinized to determine whether the request is reasonable. Within the agent, it is necessary to provide mechanisms that will protect the resource from being damaged or lost (become inaccessible) due to errors arising from the faults in the structures that implement the agent.

4. Communication Protocols

Let us return to the network in Figure 2. The communication agents deliver messages that are just strings of bits. The application agents exchange messages that represent requests to use a resource guarded by the receiving agent (requests to perform specific operations on a resource) and the corresponding replies. These messages may contain (logically) values of abstract objects meaningful at that level.* The values of these abstract objects have to be translated (encoded) into a string of bits for delivery to another node and decoded to the proper abstract objects at the receiving node.

The translated messages may have to be partitioned into packets. The messages are checksummed, so that transmission errors can be detected. In general, it is very difficult to correct transmission errors at the receiving node, since transmission errors are bursty (affect not just a single bit, but several bits). Checksum facilitates detection of errors, where the number of detectable simultaneous errors is determined by the size of the checksum field. Correction is performed through retransmission. In general, once a message has been translated into a string of bits, the communication protocols should take care of the correct transmission. However, the primary responsibility for checking that a message has been acted on, that is, ensuring that the agent that sent the message will not wait indefinitely must rest with the application. In addition, a message may contain a higher-level error: either a message has not been constructed properly by the application

---

* An object is an entity that has a unique name and a state (value) that can be modified by invoking operations defined on the object. The system provides several basic types of objects (e.g. integers, characters, arrays). Other types may be defined by choosing a set of lower level objects as a representation and providing a set of programs that implement the operations that the user of such an abstract object is allowed to invoke; these programs provide the only access to the representation of the abstract object.

agent (wrong command or wrong data) or the translation from abstract data to the bit representation has not been done correctly.

One of the most difficult problems in this type of distributed system is that unless an explicit reply (or an acknowledgement) is received, it is impossible to determine with certainty whether a message sent to an agent at a different physical node has been received and processed by that agent. The fact that no response is received from the target agent may have several different causes:

a) the message did not get to the recipient node because of a bad connection,

b) the message could not be delivered because the recipient node is down,

c) the message has not been delivered to the target agent because the node failed,

d) the reply has not been generated because the agent failed,

e) the reply has not been generated because the node failed,

f) the reply has not yet been generated because of the contention for the needed resources at the recipient node, or

g) the reply did not get back because of a bad connection.

Even if it were possible to determine the exact cause, for some of these situations (namely, d and e) it is in general, impossible to determine how far the processing of the request in the particular message has progressed. Unfortunately, it is also impossible to always determine the cause of not getting a response. Thus, the uncertainty about what happened to the request is even greater.

The only defense against possibly waiting indefinitely for a response is to use a timeout mechanism. The sender can specify a time interval after which it gives up waiting for the response; the timeout mechanism will alert

the sender when such a time interval already elapsed. The possible reactions of the sender to a timeout event can be divided into two categories: the sender decides to give up the attempt to communicate with the particular agent, or, the sender decides to resubmit the request. Because of the uncertainty discussed above, it is possible that the first request will eventually be processed. Thus, in the first case, the request may be processed in spite of the sender's decision not to continue and may conflict with the subsequent actions taken by the sender after the timeout. In the second case, the same request may be processed twice, possibly leading again to an inconsistency. Thus, in situations where an inconsistency may arise from such internode requests, it is necessary to use special (often complex) protocols [9, 11, 14, 18, 22].*

Figure 3 shows a flow of error notifications for different types of errors. In the program of a sending agent, it is necessary to provide handlers for the errors of type A and I. In addition, errors internal to the agent (arising from the agent's implementation) must be handled. A possible syntax for a send command might be:

```
send request(args) to X timeout time:
        reply1(formals) do S1;
        reply2(formals) do S2;
          .
          .
```

* This problem cannot entirely be dismissed if both communicating agents reside on the same physical node, since each agent may be implemented as a separate and independent process.

```
        failure(formals) do Sfailure;
        timeout do Stimeout;
    end;
```

Execution of this statement results in sending a message consisting of a request and some arguments to agent X. Errors of type A would be reported in one of the "reply" statements. "Failure" covers various errors that are detected by the system.

5. Availability of Resources: The Problem of Replicated Data Objects

The problem of the communication protocols discussed above is closely related to the problem of availability of resources. As discussed earlier, availability is a broad and confusing subject. Availability can be interpreted as the delay experienced when accessing a particular resource. Availability is constrained by two factors: 1) the efficiency of the system, that is, the actual physical delay and queueing time in the abstract network, and 2) failures in the abstract network.

From both of these aspects, availability can be enhanced if several instances (copies) of a resource are maintained at different physical nodes:

i. coping with failures: If some node fails, or if communication with some node fails, it should be possible for agents at other nodes to continue their work. That means that the resources provided by the failed (or inaccessible) node have to be provided by some other node(s) in the remaining operational network (each operational partition of the network).

ii. coping with bottlenecks: Even if the nodes and the communication network of a distributed system never fail, a single instance of a resource may

not provide sufficient availability. A resource may become a bottleneck; also, the communication delays, especially in a long-haul network, may be substantial, and it thus may be desirable to have a local instance of the resource (and, consequently, support multiple copies).

In the systems under consideration, the most important type of resource is information contained in various (abstract) data objects. Maintaining multiple copies of data objects that need to be frequently updated represents a special problem, as discussed below.

It is usually desired that the fact that there exists more than one copy of a specific resource is invisible to the user. That is, logically, there is only a single instance of a specific resource. The representation, however, consists of several copies at different physical nodes. To present such a view to the user, the copies must be kept identical. That is, for the case of mutable objects, if the logical object is modified, all copies have to be modified. This can be performed in several different ways. Figure 4 shows a centralized scheme, where one of the copies is designated the master, and the other copies are backup copies. All requests are always channeled to the master copy, or, better, to the agent associated with the master copy. If the master copy is modified, its agent supervises distribution of the changes to the backup copies. When the master copy becomes unavailable, one of the backup copies will become the new master. This type of scheme was described by Alsberg [1]. This scheme, however, increases availability only from the point of view of failures; the master copy is, of course, still a bottleneck. To take a full advantage of the multiple copies, it should be possible to use any copy for both reads and updates. However, consider the situation depicted in Figure 5 where two users request to change the same object. If the two requests are performed simultaneously and independently, after the changes

made as a result of these requests have been distributed to the other copies, some or all of the copies may end up with a wrong value. Thus, such distributed updates have to be carefully scheduled (synchronized) [6, 20, 23]. However, synchronization in this type of environment represents a substantial overhead [8]. In addition, synchronization becomes very complicated due to possible failures of the nodes and the communication network. Thus, the attempt to enhance availability through multiple copies may be completely negated by the delay encountered due to the synchronization problem.

The preceding discussion was presented here to demonstrate both how reliability and performance are interdependent, and that it is important to understand clearly the purpose of supporting multiple copies. To cope with bottlenecks, it may not always be necessary to have completely identical copies. That is, this problem can be handled by allowing multiple versions of an object. A mechanism for maintaining multiple versions of objects in such a way that a consistent version of a set of objects can always be obtained was developed by Reed [18]. In addition, Reed's mechanism solves the problem of updates and backout in a distributed system in a most natural way. However, to prevent loss of information, the most current version ought to have at least one backup copy. The scheme that combines multiple versions and backup copies is sketched in Figure 6. The agent associated with the master copy is responsible for performing all the modifications for concurrent requests. Basically, each modification request creates a new version of the object. Each new version is distributed to the backup agents in a strictly sequential manner.

6. Use of Replication for Recovery

The preceding section discussed the problem of maintaining multiple copies of mutable data objects. This section presents a scheme that uses multiple copies of data objects to recover from node failures and user errors.

At each level,* the information processing system supports a set of operations that from the point of their user ought to be atomic. An atomic operation is indivisible in the following sense: 1) the result of an atomic operation performed by one process is not affected by other, concurrent processes, and 2) such an operation is either carried to its completion, according to its specification, or, if it is aborted, it leaves the system in the state it was prior to the invocation of that operation. In the underlying implementation, such an operation may be composed of many different lower level operations (actions). Thus, to make an operation atomic, it is necessary to ensure that the lower level operations not only are synchronized properly, but that their effects do not become permanent until it is clear that the entire operation in question will complete successfully. This can be handled either by performing the permanent changes dictated by that operation only upon its completion or by maintaining enough information about these changes such that they can be undone if the operation cannot complete normally. At the time the changes become visible outside of the operation that caused them, the operation is committed.

Much high quality research has been done on the subject of atomic operations (frequently called transactions or atomic actions) [7, 13], especially in the context of distributed systems [9, 11, 14, 18, 22]. In the

* Thus far, we have talked about two levels: the application level and the system level. However, each of these "levels" may be in reality a hierarchy of several levels, each level presenting a well defined interface to the level built immediately on the top of it.

schemes for distributed systems, so called two-phase commit protocol is used to coordinate changes made at different physical nodes. This type of protocol is based primarily on backward recovery: until it is agreed that an operation can be run to completion, a failure or inaccessability of one of the involved nodes will eventually cause the operation to be terminated and its effects undone. If copies of the needed resources requested from the failed or inaccessible nodes are available elsewhere in the network, it may be possible to restart the operation, using a different set of nodes. Now, in this process, a substantial amount of work may be lost, depending on when the failure that triggered the backward recovery has occurred. For complex and expensive operations it seems desirable to have a forward recovery scheme that allows the operation to continue by switching to another copy of a resource as needed. Backward recovery, however, is still necessary, in particular to deal with human errors, synchronization errors (two or more operations may end up in a deadlock) and residual algorithmic faults (software bugs).*

The changes made to an object ultimately must be propagated to all copies. However, since some of the copies may be temporarily unavailable, it is not feasible to wait until all copies have the new value before an operation is deemed to have completed. The n-resiliency protocol developed by Alsberg [1] requires that only n copies must be updated before a successful end of operation is assumed. The protocol ensures that the changes do eventually propagate to the rest of the backup copies. Some problems, however, may occur if the network becomes partitioned and the partitions are to be merged later. To be able to cope with an arbitrary partitioning of the

* It is useless to switch to another copy of a software resource if the failure was caused by an algorithmic fault. However, after a backward recovery from such a fault, a different algorithm can be tried [16].

network, n ought to be more than half of the number of the existing copies, that is, a majority of the copies must be updated to complete an operation on a replicated object. If there are k copies in the system then n = [(k+1)/2]. Let us call this last scheme a majority commit protocol. Finally, it is assumed that each copy of an object has its own agent; these agents are the entities that implement the desired protocols.

For backward recovery, replication of mutable data objects could be used in the following way:

Scheme A: i. Perform the set of actions on the master copy as requested.

ii. To commit the operation, use the majority commit protocol.

iii. To abort the operation, undo changes in the master copy by reading in one of the backup copies.

Scheme A is straightforward, however, it has several drawbacks. First, the commit step may take an indefinitely long time if it is impossible to obtain a majority of votes. Second, it precludes forward recovery, since the backup copies do not have any information about the individual actions performed on the master. Third, it precludes concurrent use of the object by two or more operations since 1) the master copy in addition to containing the changes made by an operation that is about to be completed would also contain changes made by operations still in progress, and 2) if an operation needed to be backed out, all changes (that is, also the changes made by other operations) made since the last time backup copies were updated would be lost.

To deal with the first problem, it must be possible to abort the operation if a majority of votes cannot be secured within a reasonable time period. Thus, a two-phase commit protocol is needed here too. In the first phase, changes are distributed to the backup agents. If a majority of the agents confirm that they have received the new version (since the master agent

already knows the new version, only [k/2] backup agents need to respond), the master agent sends a commit request to all backup agents and performs the changes in the master copy. Concurrency is a much more complex problem: different types of objects allow different degrees of concurrency. For example, if the object in question is a file, and the individually lockable entities are records, then if the sets of records used by two operations are mutually exclusive, or if those records that appear in both operations are not only read but not updated, the two operations can be performed concurrently.* Assuming that concurrency is carefully controlled, Scheme B facilitates backup and recovery of each operation individu

Scheme B: i. In the master copy agent, construct a list of changes to be made to the object.

ii. To commit the operation, use the two-phase majority commit protocol.

iii. To abort the operation, delete the list of changes at the master copy agent.

Finally, another refinement is needed to support forward recovery:

Scheme C: i. In the master copy agent, construct a list of changes to be made to the object; each time a new action is requested on an object, send the request and the current list of changes to all backup agents using the majority commit protocol.

ii. To commit the operation, send the final list of changes and perform the changes using the two-phase majority commit protocol.

iii. To abort the operation, delete the list of changes in all copies.

---

* For a more rigorous treatment of this subject see [7].

iv. To continue the operation with a new master, repeat failed (last) action at the new master, continue as described in part i.

7. Conclusion

Distributed systems present a number of new problems that have to be solved to achieve reliable operation. The major problem is uncertainty, the fact that it is impossible to always know the entire global state of the system. Also, in distributed systems the tradeoff between reliability and performance becomes much more prominent than in centralized systems.

This paper presented distributed applications as a network of agents that represent various resources and that communicate by sending messages. The reliability problem was addressed along two lines: the replication of the resources and the communication between agents. Work is continuing in both of these directions. In the area of communication protocols, it is necessary to work out the details of how the three types of mechanisms, error messages, exception signalling and timeouts fit together. The multiple copy problem needs to be analyzed from two angles: the robustness of the proposed scheme, that is, the types of faults it can tolerate, and performance. Unless this type of scheme can be made reasonably efficient, it will be of little value in building actual systems.

REFERENCES

[1] Alsberg, P.A., "A Principle for Resilient Sharing of Distributed Resources," Proc. of the International Conference on Software Engineering, San Francisco, California, October, 1976, pp. 562-570.

[2] Avizienis, A., et al., "The STAR (Self-Testing and Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," IEEE Trans. on Computers, C-20, 11, November, 1971, pp. 1312-1321.

[3] Bartlett, J.F., "A 'Non-Stop' Operating System" Tandem Computers Inc., Cupertino, California, 1977.

[4] Baskin, H.B., et al., "PRIME - A Modular Architecture for Terminal-Oriented Systems," Proc. AFIPS SJCC, 1972, pp. 431-437.

[5] Clark, D.D., et al., "An Introduction to Local Area Networks," Proc. of the IEEE, 66, 11, November, 1978, pp. 1497-1517.

[6] Ellis, C.A., "Consistency and Correctness of Duplicate Database Systems," Proc. of the Sixth Symposium on Operating Systems Principles, November, 1977. pp. 67-84.

[7] Eswaren, K.P., et al., "The Notions of Consistency and Predicate Locks in a Database System," Comm. of the ACM, 19, 11, November, 1976. pp. 624-633.

[8] Garcia-Molina, H., "Performance Comparison of Update Algorithms for Distributed Databases," Stanford University Digital Systems Laboratory, Technical Note No. 143, June, 1978.

[9] Gray, J.N., "Notes on Data Base Operating Systems," Lecture Notes in Computer Science, 60, Springer-Verlag, 1978, pp. 393-481.

[10] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," Comm. of the ACM, 17, 10, October, 1974, pp. 549-557.

[11] Lampson, B., et al., "Crash Recovery in a Distributed Data Storage System," Xerox Palo Alto Research Center, 1976, (to appear in Comm. of ACM).

[12] Liskov, B., et al., "Abstraction Mechanisms in CLU," Comm. of the ACM, 20, 8, August, 1977, pp. 564-576.

[13] Lomet, D.B., Process Structuring, Synchronization, and Recovery Using Atomic Actions," Proc. of an ACM Conference of Language Design for Reliable Software, ACM Operating Systems Review, 11, 2, April, 1977, pp. 128-137.

[14] Montgomery, W.A., "Robust Concurrency Control for a Distributed Information System," M.I.T. Laboratory for Computer Science, Technical Report No. 207, December, 1978.

[15] Ornstein, S.M., et al., "Pluribus - A Reliable Multiprocessor," Proc. AFIPS NCC, 1975, pp. 551-559.

[16] Randell, B., "System Structure for Software Fault Tolerance," IEEE Trans. on Software Engineering, SE-1, 2, June, 1975, pp. 220-232.

[17] Randell, B., et al., "Reliability Issues in Computing System Design," Computing Surveys, 10, 2, June, 1978, pp. 123-165.

[18] Reed, D.P., "Naming and Synchronization in a Decentralized Computer System," M.I.T. Laboratory for Computer Science, Technical Report No. 205, September, 1978.

[19] Roberts, L.G., et al., "Computer Network Development to Achieve Resource Sharing," Proc. AFIPS SJCC, 1970, pp. 543-549.

[20] Rothnie, J.B., et al., "The Redundant Update Methodology of SDD-1: A System for Distributed Databases," Computer Corporation of America, Report CCA-77-02, February, 1977.

[21] Svobodova, L., et al., "Semantics of Distributed Computing," Progress Report of the Distributed Systems Group, M.I.T. Laboratory for Computer Science, September, 1978.

[22] Takagi, A., "Concurrent and Reliable Updates of Distributed Databases," M.I.T. Laboratory for Computer Science, Computer Systems Research Division, Request for Comments No. 167, November, 1978.

[23] Thomas, R.H., "A Solution to the Update Problem for Multiple Copy Data Bases Which Use Distributed Control," Bolt Beranek and Newman, Report No. 3340, July, 1976.

[24] Wensley, J.H., et al., "The Design, Analysis, and Verification of the SIFT Fault Tolerant Software," Proc. International Conference on Software Engineering, San Francisco, California, October, 1976, pp. 458-468.

[25] Wulf, W.A., et al., "An Introduction to the Construction and Verification of Alphard Programs," IEEE Trans. on Software Engineering, SE-2, 4, December, 1976, pp. 253-265.

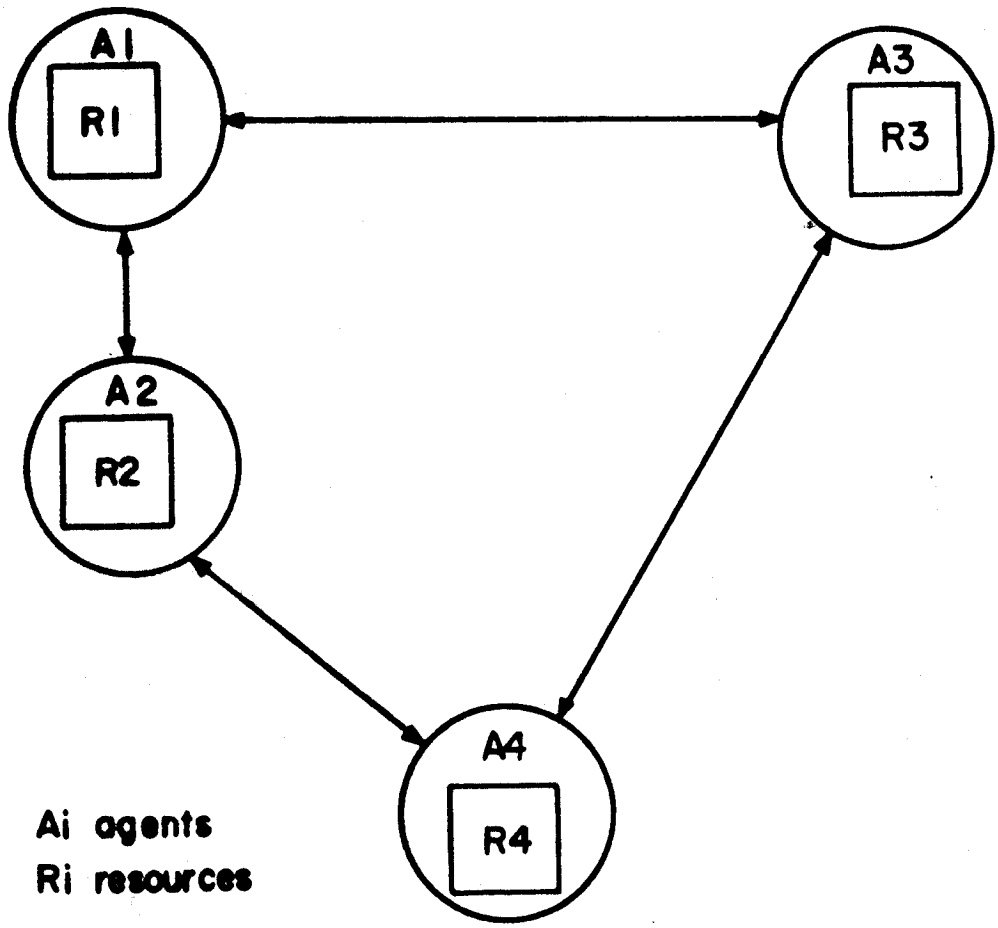
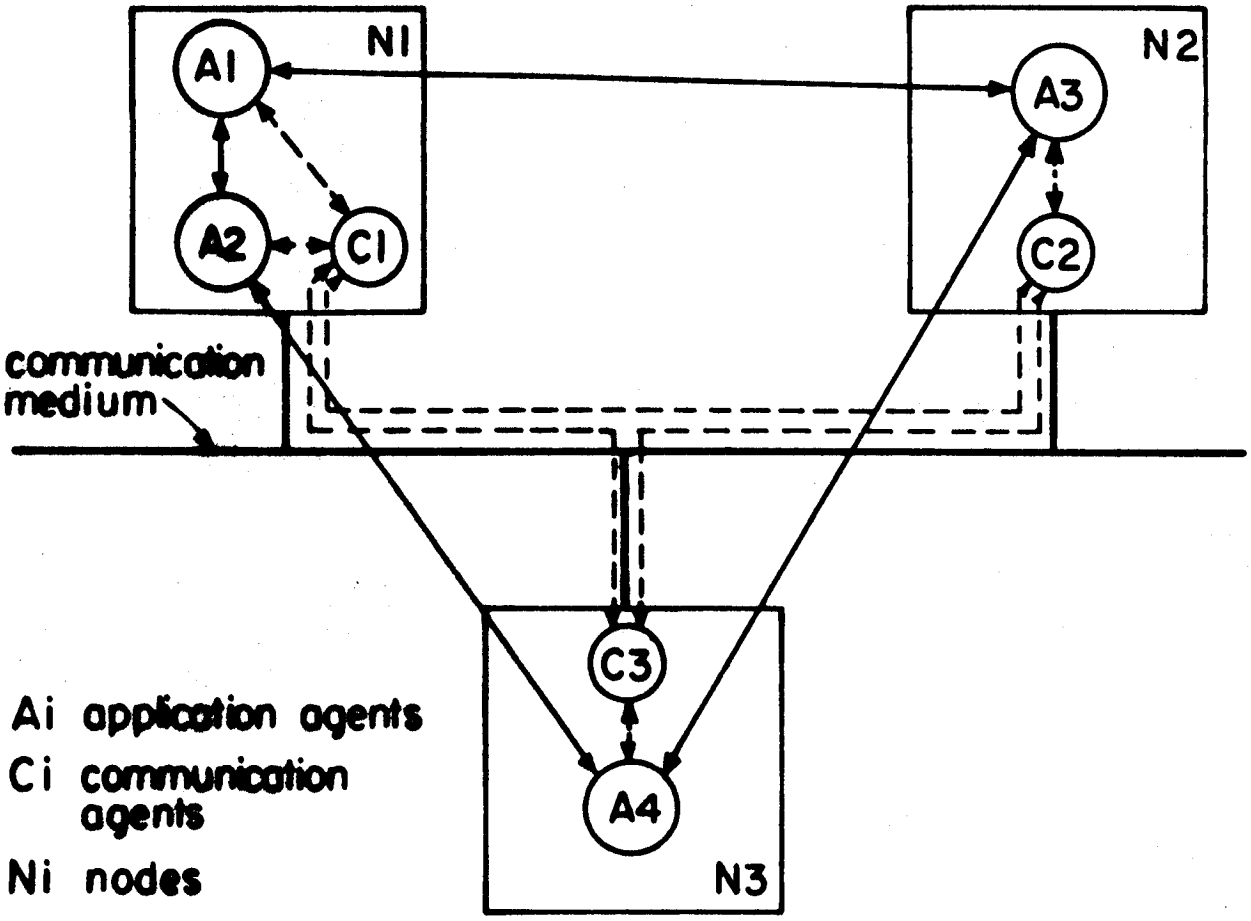


Figure 1: A model of a distributed application.



↔ possible communications on the application level

↔ flow of information in the network

Figure 2: The abstract network: communications on and between the application and the system levels.

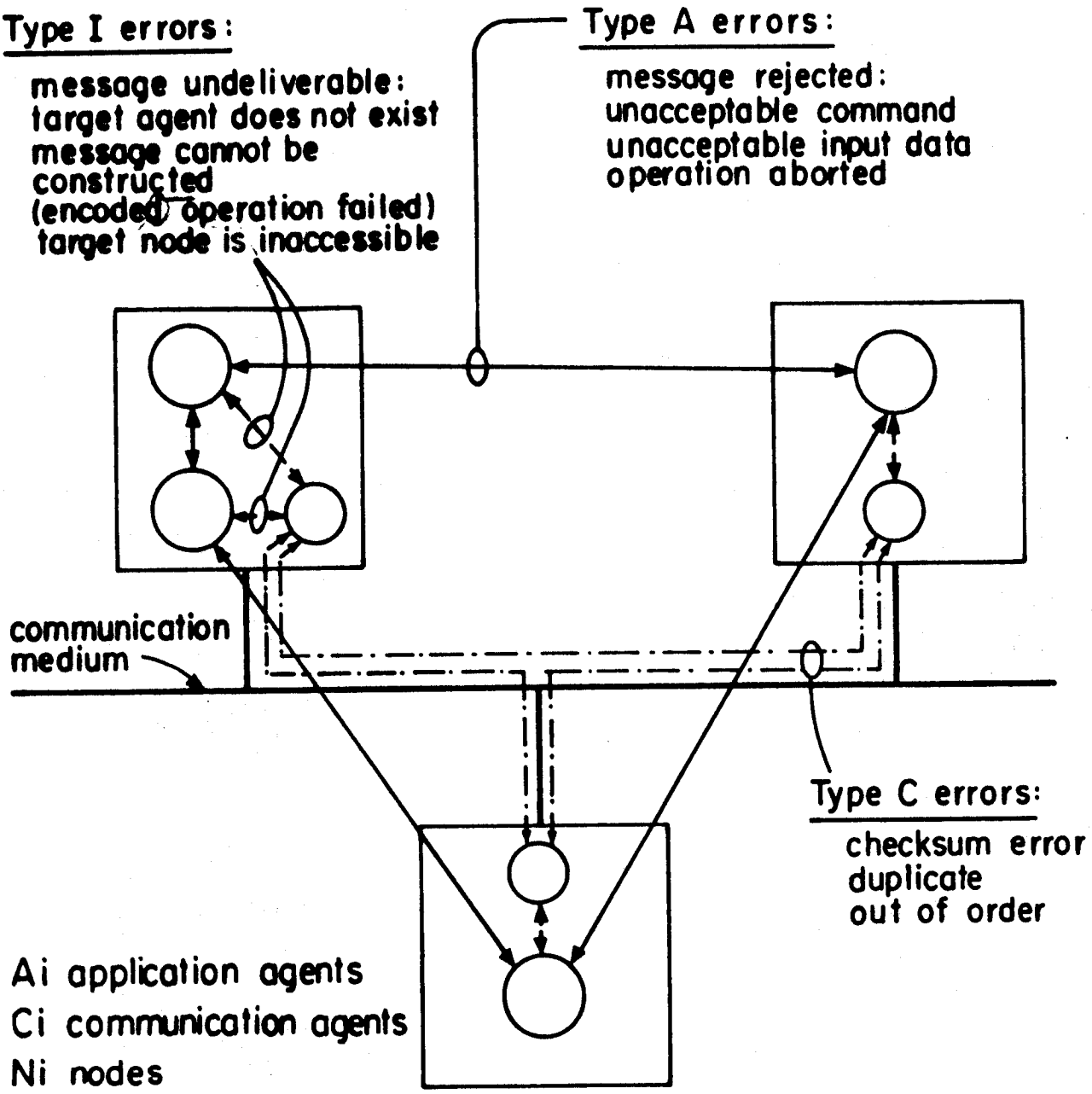
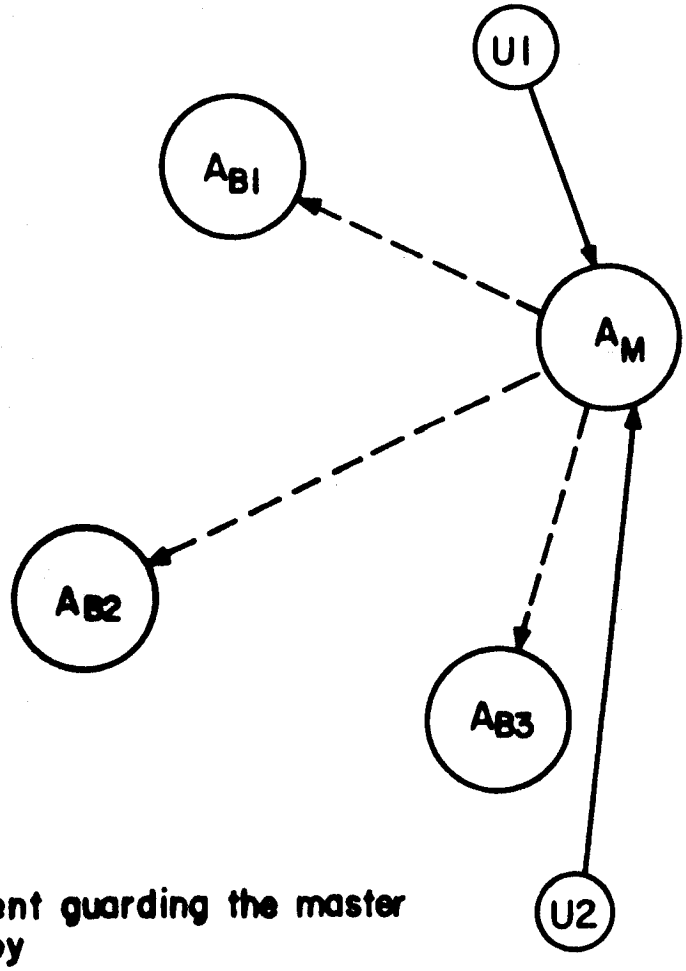


Figure 3: Flow or error notifications in the abstract network.



A_M agent guarding the master copy

A_{Bi} agents guarding the backup copies

Ui users

⟶ update requests

⇢ propagation of changes

Figure 4: Centralized update scheme (master/backup).

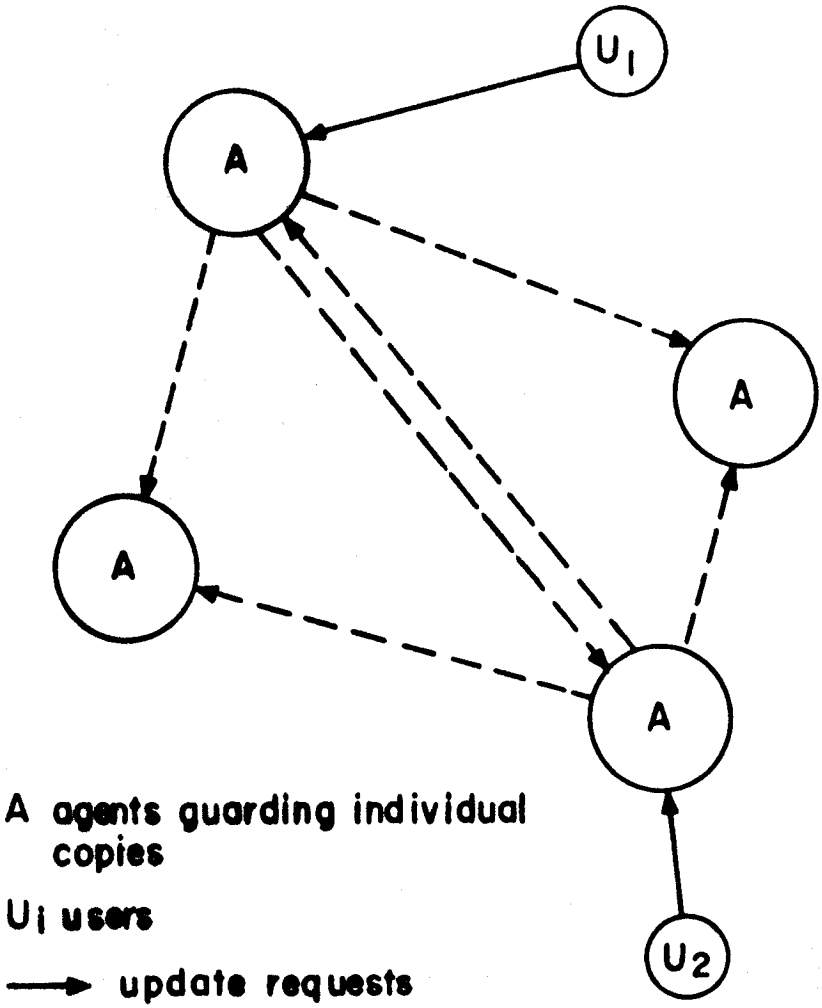
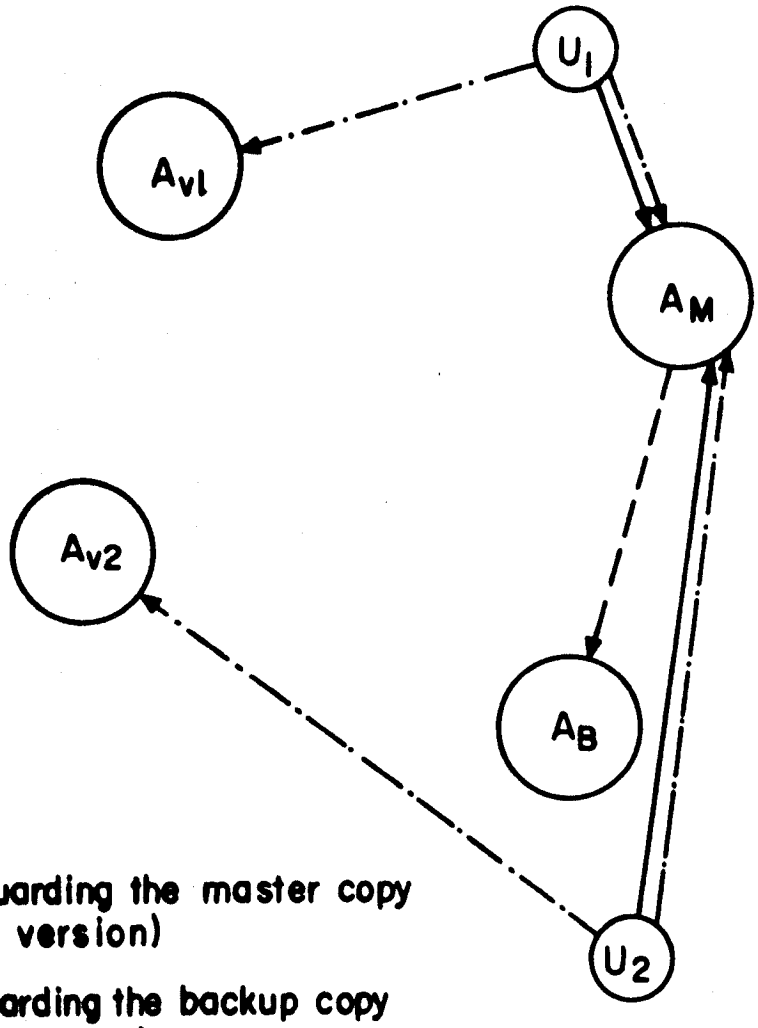


Figure 5: Distributed update scheme.



A_M agent guarding the master copy (current version)

A_B agent guarding the backup copy (current version)

A_{vi} agent guarding version vi (current or older version)

Ui users

——→ update requests

— → propagation of changes

—·—→ read requests

Figure 6: Multiple version scheme with backup.