MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Laboratory for Computer Science

Computation Structures Group Memo 171

Semantics of Distributed Computing
Progress Report of the Distributed Systems Group
1977 - 1978

by

D. G. Clark
I. Greif
B. Liskov
L. Svobodova

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

October 1978

SEMANTICS OF DISTRIBUTED COMPUTING

PROGRESS REPORT OF THE DISTRIBUTED SYSTEMS GROUP

September 20, 1978

D.D. Clark, I. Greif, B. Liskov, L. Svobodova

1. Introduction

Computer systems should reflect the structure and needs of the problem to which they are being applied. For many applications, a distributed computer system represents a natural realization. For both technical and economic reasons, it is likely that for many existing applications, distributed computer systems will replace conventional computer systems built around a large central processor, and that new applications will emerge based on distributed information processing. However, before such systems are feasible a better understanding of how to construct them is needed. Our project is aimed at providing this understanding.

The move towards distributed processing has become feasible mainly because of the rapidly dropping cost of computer hardware and the increasing power and flexibility of mini and microcomputers. The move toward distributed systems will be dictated, however, by their "naturalness", and by the many technical advantages they offer over centralized systems. These advantages include the following:

Availability. Availability of information can be increased by replicating it at several nodes. This arrangement not only increases the access bandwidth to the information, but in case of a failure of one of the nodes or some communication link, the information remains accessible.

appearance of a coherent system. The project discussed in this report is to develop an integrated programming language and operating system to support well-structured design and implementation of distributed applications.

1.1 Distributed Systems of Interest

The area of "distributed systems" has become a popular source of systems research projects. It has also become an important term in marketing computer equipment. Unfortunately, because of this popularity, the terms "distributed systems" and "distributed processing" are frequently misused, often referring to such conventional concepts as remote job entry, use of terminal concentrators, or multiprocessor organizations.

The distributed systems considered in our project can be described loosely as organizations of highly autonomous information processing modules, called nodes, which cooperate in a manner that produces an image of a coherent system on a certain defined level. Autonomy is the key characteristic that eliminates most multiprocessor organizations from this class of distributed systems. Certainly, a distributed system has more than one processor, since it has at least one processor in each node. However, in a distributed system, the nodes are highly independent, each having its own primary memory, possibly even some secondary storage, and its own interface through which it communicates with its environment (e.g. user terminals, sensors). The individual nodes are connected by a communication network; the communication delay may be highly variable and unpredictable. The communication network might be a long-haul network such as the ARPANET [15], a local area network [2], or a suitable combination of these two types. Each node has access to its own memory only; that is, inter-node communication is possible only by explicitly exchanging messages, not through shared memory. Finally, physical

Distributed systems have only lately become a focus of programming language research. In the past, programming languages have mostly not addressed concurrent programs. More recent languages (e.g. Concurrent Pascal [1]) Modula [22]) have had features for concurrency, but within the context of a single processor: these languages are based on the assumption that programs interact through shared memory, which is not consistent with the concept of autonomous nodes with private memory. There is related work at Oxford [9], the University of Rochester [6] and at MIT [4,7], but this work does not place strong emphasis on integrating the language and operating system features.

Indeed, we feel that our emphasis on integration of language and system is a key factor in our work and distinguishes it from other related work. Much of what distributed programs do falls into what is usually considered to be the systems area, including such topics as synchronization of access to shared information, and protection. However, programs are written in a programming language, and proper primitives in that language can greatly influence the structure of programs. By integrating the two areas we expect to achieve a greater impact on the construction of distributed systems than could be accomplished in either area separately.

2. Study of Applications

It is essential that the mechanisms we develop to support construction of distributed applications will cover the real distributed processing problems. To this end, we have studied a number of applications, both by direct observation [19,20] and by surveying related work as discussed earlier. This study was hampered by the lack of existing distributed systems; for example, banking systems are not yet distributed, although a distributed system is

not possible to restrict in advance the modes of sharing among users. It is necessary to communicate both data and programs, but from the point of view of the mechanics of the actual exchange of information this type of system could be included in the first category.

The distribution can take place along two main lines, based on functional separability or on the non-uniform distribution of the use of databases. Functional distribution means that different nodes support different services. Such systems seem natural for control of industrial processes, where different nodes control different parts of a process, or in such systems as aircraft, where different nodes process information from different sensors. However, this approach seems to be also advantageous in service sectors such as banking [19].

Another category of distributed systems is a system where an individual processor supports the same services but on a different part of a database. A typical example is a bank with many branch offices. Each branch has its local accounts, but it should be able to serve a bank's customer whose account is at another branch. Since such remote requests are much less frequent than manipulation of the local accounts, partitioning of the bank's accounts database (that is, maintaining accounts on a computer at their local branch) is a natural approach.

It must be said that the division between functional distribution and database distribution is not clean; in most cases, a distributed system will to some extent include both. The latter case, however, implies an integrated database, while in the former case (functional distribution) the databases used by individual servers are much more independent. In some ways, the functional distribution is a more general case. A distributed database

is a large gap between the application and the low level communication protocols. Usually, this gap results in a rather ad hoc implementation of the application.

Our target is an intermediate level, called the programming system, which will support a well-structured design, implementation, maintenance and control of distributed applications. This level is more than a programming language in a traditional sense. Rather, this level is envisioned as a set of tools that include primitives found in conventional higher level languages such as Pascal or PL/1, but also primitives normally assumed to be a part of an operating system, for example, long-term storage and cataloging of information or control of protection safeguards. Thus, this programming level will integrate the programming language and the operating system. More strongly, this level will integrate a programming language and a distributed operating system.

The design goals for the programming system include:

- Aim for as high a level as possible, but application independent. Our system is intended to be used to implement many diverse applications, for example, both command and control systems and administrative systems like inventory control systems. To adequately support such a class of applications, the language should be as high level as possible but general purpose. One need that all applications share is the ability to exchange potentially quite sophisticated messages.
- Support well-structured programming. Since our primary motivation is to ease the task of the application programmer, we feel that the embedded language should borrow from existing language work, in particular building on languages such as CLU [11] and Alphard [23],

the processing that is needed to translate an object in memory into a message transportable by the communication network and vice versa: the translation is accomplished using special operations of the object's type. Note that this translation is always needed; a language that requires messages to be composed of low level objects simply obscures this fact.

- Allow explicit control of the application distribution.

Conceptually, the target level can be viewed as an abstract network of processes where application-defined processes communicate via messages that contain high level commands, data and responses. In an ideal situation, this is all that would need to be seen by the application programmer. However, underneath this abstract network is the set of physical nodes and the communication lines that connect them. Our study of applications has indicated that the mapping of the objects used by an application into the physical set of nodes has to be made visible to the application programmer. We are also assuming that objects do not move dynamically from node to node, depending on the degree of demand (such dynamic migration is often assumed in the "distributed" systems consisting of many, relatively tightly coupled, mini or microprocessors). Rather, when a specific node is chosen to be the (new) home of a particular object, an installation of the object has to be explicitly requested using commands provided by the programming system. This assumption is based on the belief, discussed earlier, that such placement decisions will often be based on non-technical factors external to the system [3].

although in some environments (such as LCS) there is often little distinction between the two classes of users. Also, it should not be necessary for all nodes in the distributed system to support the full language; each node need only support the appropriate (high level) internode communication protocol.

4. Entities

In this section we discuss the universe of entities (e.g. programs, data) that take part in a distributed computation. We are not concerned with all aspects of the behavior of the entities, but rather limit our attention to questions concerning the locations of entities within the network and the possible relationships among the entities. We assume that each entity has an _identity_ that is permanent; an entity can be referred to by giving its _name_.

4.1 Location of Entities

The universe of entities is spread across the physical nodes that make up the network. One question that arises concerns the location of entities: is an entity permanently located at a particular node, or can it move from node to node?

To make a decision here, we must consider several issues:

1. Earlier we discussed our conclusion, based on an analysis of applications, that the application programmers must be able to control the location of entities. Note that, at the least, this conclusion precludes automatic relocation of entities by the system, although relocation under program control would still be possible.
2. We are assuming that nodes are autonomous and possibly heterogeneous. Even under program control it is possible to move an entity to an autonomous node only if that node is willing to accept it. Furthermore, if that node is different from the current home

Objects have a state (value) that may change. If the state can change during the object's lifetime, then the object is mutable.

A process can communicate with another process by sending it a message. We assume that the syntax and semantics of message passing is independent of the nodes of residence of the two communicating processes (although certain optimizations can be performed by the system if both processes reside at the same node). A process can use an object by performing (invoking) an operation on it (or by invoking it if it is a procedure); again, the semantics of invocation is the same regardless of the nodes of residence.

We have just described a model in which there are two basic primitives: invocation and message passing. We intend that the semantics of invocation is distinct from message passing: the primitives are really different. (We expect that these two primitives will also be distinguished syntactically, but that is a separate decision.)

If an actor-like view is taken, there is only one basic primitive, message passing, so our model seems more complicated. However, we believe that it is more natural than the actor model and will therefore be easier for programmers to understand. If programs built out of actors are examined, it is clear that there are "data-like" actors, "procedure-like" actors and "process-like" actors. We believe these differences are fundamental and should be reflected in the language and its semantics.

4.3 Restrictions on Referring to Entities

Now we address the subject of entities referring to entities. An entity may refer to another entity by using or containing its name. For example, a process will have local variables that may contain the names of other entities (both processes and objects); as the process executes, it can use these names.

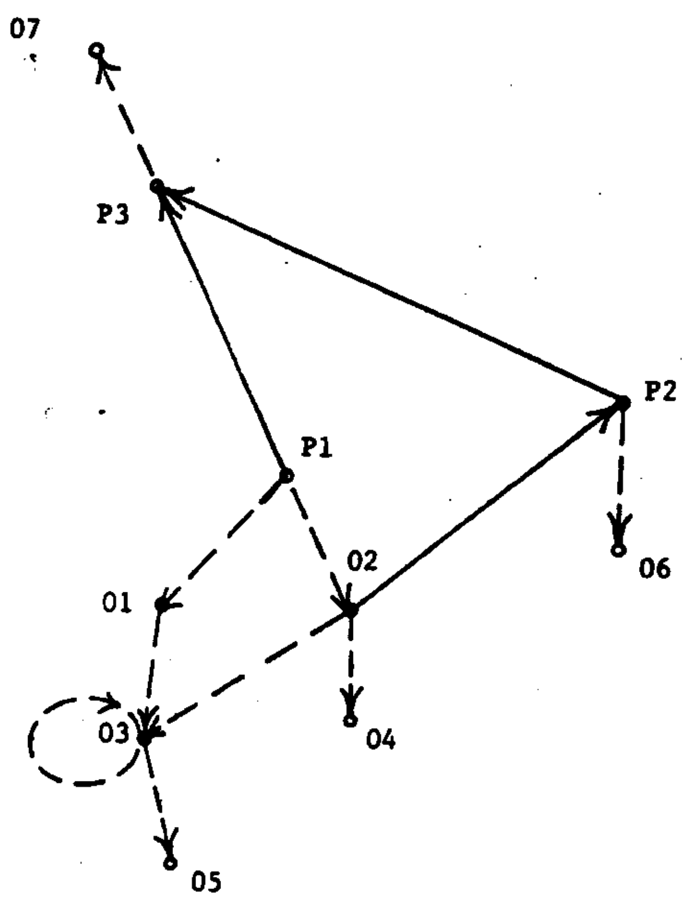


Figure 1: Example of possible relationship of processes and objects.

A guardian should not be assumed to know a priori about all processes that may request operations on the guarded objects. Furthermore, if a process requests an operation on data that are available only through the guardian, such a request may fail, since the guardian may refuse to release requested data, or in some cases may even destroy the data at its own discretion.

The abstract network model requires two extensions to be useful. First, the requirement that local address spaces of processes are disjoint may need to be relaxed. To obtain sufficient parallelism, it will probably be necessary to support complicated guardians consisting of several processes that share objects. This could be accomplished by a special syntactic construct, something like a serializer [8], that defines the processes making up the guardian and their intercommunication; all the processes in the guardian would reside at the same node.

Second, in the case of a guardian that guards several objects, some efficient mechanism is needed that permits a user process to specify to the guardian the particular object of interest, and for the guardian to determine that the object so specified is one it guards. The system provides no guarantee, however, that such an object continues to exist as long as the user can specify it.

5. Reliability Issues

Reliability is and will be one of the major issues in information processing systems. As discussed earlier, distributed systems provide a potential for enhanced reliability; however, this potential needs to be exploited through proper design. This section discusses the reliability problems in distributed systems and the mechanisms needed to achieve reliable operation of a distributed application.

left to the application level.* Thus, the system ought to provide sufficient mechanisms for masking certain classes of failures arising from the operation of the hardware and the software that supports the application programs. However, the system also has to provide suitable language constructs for the application programmer to facilitate handling of the application specific failures and communication of the system detected failures to the application programs.

5.1 Communication Protocols

The abstract network is supported by a physical network of nodes and communication lines. Figure 2 shows the abstract network mapped into the physical network and the communication processes that control the physical delivery of messages among the nodes. The application processes exchange messages that, logically, contain values of high level (abstract) objects meaningful at that level. The values of these 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. At the system level, messages, now in the form of a string of bits, may have to be partitioned into packets. The messages are checksumed, so that transmission errors can be detected. It is difficult to <u>correct</u> 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

---

* In the class of system level failures, there is a gray area where a decision has to be made as to whether these failures will be masked by the system level or reported to the application level.

should take care of the correct transmission. However, the primary responsibility for checking that a message has been acted on, that is, ensuring that a process that sent a message will not wait indefinitely, and also that the message contains values acceptable from the application standpoint, must rest with the application.

The language constructs needed to permit an application process to deal with failures of another application process with which it is communicating or attempting to communicate, to defend itself from improper use, and to deal with the failures in the system level are discussed in Section 6. This section concentrates on the system level.

A truly reliable system level should be prepared to deal not just with communication errors that result in a loss or garbling of messages sent across a physical communication link. A reliable system level should not lose messages that have been presented to it by the application processes and queued for delivery. That is, the message queues should be recoverable in case of a physical failure of a node. This requirement becomes very important if translation from an abstract data object to the corresponding bit representation is a costly operation, or if the input to such a translation step is not automatically repeated (e.g. message typed by a user). This argument can be extended to the requirement that the system should guarantee delivery of all messages it has accepted from the application processes. That means that in addition to providing recoverable queues for messages that have not been sent yet, the system must continue trying to send the queued messages until it eventually succeeds. At the receiving node, the messages have to be stored again in recoverable queues, until they are picked up by the target application process.

detected deadlock or because of a failure of some entity it uses). Redundant copies make it possible to restore the current state or to backup some earlier state of an object.

The issues regarding the reliability of individual objects are not specific to a distributed system; <u>any</u> information processing system should support backup and recovery of stored objects. Distributed systems, however, can increase the availability of information and services. "Availability" can be interpreted as the delay experienced when accessing a particular object. This definition has two connotations: one is the efficiency of the system, that is, the actual physical delay and queuing time in the abstract network (case E); the other source of delays are the failures in the abstract network, that is, the reliability aspects (case R). Redundancy is used for both of these subcases:

<u>Case R</u>: If some particular node or communication with a particular node fails, it should be possible for the other nodes to continue their work. Since the failed (or inaccessible) node may contain objects needed by the other nodes, to increase availability means to maintain several copies of shared (shareable) objects on different nodes.

<u>Case E</u>: Even if the system never fails, a single copy may not provide sufficient availability. A single copy of information or service 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 copy (and, consequently, support multiple copies).

The question that needs to be answered is to what extent the individual copies have to be mutually consistent. It is important to distinguish between

multiple versions of selected objects, where the most current version is backed up on the system level.

6. Language Constructs for Sending and Receiving Messages

An important issue in designing a language for distributed systems is how the language recognizes pairing of messages. The basic scenario in the abstract network is one process sending a message to another process requesting some action; later there should be another message, flowing in the other direction, indicating the result of the action. It must be possible to express in the language that the two messages are related. In addition, it is necessary to address the problem that the reply may never arrive, or that the request message cannot be sent. Several approaches are possible that differ in how long (for what event) the sending process must wait before it can proceed. Closely related to this degree of waiting is what kind of failures are detectable as part of the send command.

6.1 The Waiting Approach

In this approach, the sending process is forced to wait until the response comes back from the receiver, or some timeout or failure results. A possible syntax might be:

```
send C(args) to A timeout time:
        R1(formals) do S1;
        R2(formals) do S2;
         ...
        failure (formals) do Sfailure;
        timeout do Stimeout;
        end;
```

A different kind of "send" command is needed in the receiving process, since the receiving process must be able to respond to the command without waiting for the original sender process to respond back. To receive messages, A might use a construct:

```
command case
    ...
    C(formals) do...reply R(args);...;
    ...
    end;
```

Here, A is waiting for one of a number of messages; if several are available, one is selected in a fair way. The message is then decoded, the contained data assigned to the formats, and the statements associated with the selected message are executed. The reply command sends a message back to the process that sent the message. Another form of reply:

```
    reply R(args) to B
```

which explicitly names the process to reply to will probably also be needed. (This would permit a third process to be the replier to the original sender.)

The approach sketched above has the obvious advantage of pairing sends and receives. It also has some obvious disadvantages. For one thing, there are two send commands. More important, however, is the loss of parallelism. If the sending process had other tasks to do while its request was being processed, it must either not do them, thus reducing efficiency, or it must spawn another process to do these tasks. Thus a language supporting this approach must provide rich facilities for parallelism.*

---

* Note: this is not the only reason for which such facilities for parallelism might be needed. See the discussion of guardians in Section 4.

communicates with several other processes. The port scheme could be further extended to allow the programmer to use a special port for replies indicating a failure:

send C(args) to A reply-to P failure-to F

The port F could be viewed as an entry to the "complaint department" of the respective application process.

The no-wait approach permits parallelism and is more flexible, especially in connection with ports. However, the linguistic mechanisms needed to enable the programmer to do the matching introduce extra complexity; how much flexibility is gained and how much complexity is added requires further study. The no-wait approach does not eliminate the need for supporting timeout, but now the timeout is specified at the point where the process must wait for the reply.

6.3 The In-Between Approach

This approach again makes the sender wait, but instead of waiting for the reply from the target process, the sender must wait only for some indication about the progress in the delivery of the message. For example, in Hoare's language [9], the sender waits until the replier receives the message.

The first question to ask is: does this approach offer the programmer any advantages over the other two approaches? Since sends and replies are not explicitely paired, from this point of view the in-between approach offers similar advantages and disadvantages as the no-wait approach. What is gained over the no-wait approach is that certain failures, for example, (c) and (d), or possibly even (e) can be treated as exceptions of the send command. More importantly, the completion of the send command indicates that a meaningful message (to some extent) has been received, and, if the buffer into which the

7. Protection Issues

In a distributed system, the protection problem can be simplified if we distinguish between inter-node and intra-node protection mechanisms. In the class of distributed systems considered in our project, a likely case is that a particular node is utilized by one user or at most by a set of cooperating and mutually trusting users. In this case, intra-node mechanisms are not required to have power sufficient to protect against subversion and malice. This is in strong contrast to a system such as Multics [17], and many other time-shared and multiprogrammed systems that were designed to operate properly with a set of mutually hostile users. The protection mechanism required in a single node is that which protects adequately against error and forgetfulness. This latter problem, while less severe than the problem that results from fully suspicious cooperation, is still not trivial. Presumably, the programmer must be provided a means of partitioning his computations, so that certain objects are accessible only in certain computations. This mechanism will allow him to debug new versions of software without running the risk of destroying existing objects.

We propose that a capability mechanism be the mechanism to provide this intra-node protection. By capability we mean an unforgable identifier for an object, which identifies the type of the object.* It must be presented as part of addressing an object. By constraining a procedure to execute with a limited collection of capabilities, it is easy to guarantee that the procedure will not do arbitrary damage to stored information.

---

* "Capability" is often used to mean more than an unforgeable identifier: a capability may also include a specification of the access rights, that is, a specification of which of the operations defined for the type of the object in question are actually allowed on that specific object. However, access control can also be achieved by making the object appear to be of the type that imposes the desired restrictions.

model of protection generally turns out to be that based on access control lists. While capabilities are often used in the real-world, the most obvious example being keys, the drawbacks are well known. Keys are subject to unauthorized duplication, loss, theft, etc. More relevantly, capabilities (or keys) do not provide a means to support accountability.

Both inter-node and intra-node protection requires partitioning of computations into non-overlaping access domains. The abstract network derived in Section 4 already provides such partitioning: each "node" in this abstract network has its own local address space inaccessible to other nodes. The decisions about intra-node and inter-node protection mechanisms can be extended to the abstract network: specifically, capability mechanisms will be used inside an abstract node, while access control lists will be employed for inter-node communication.

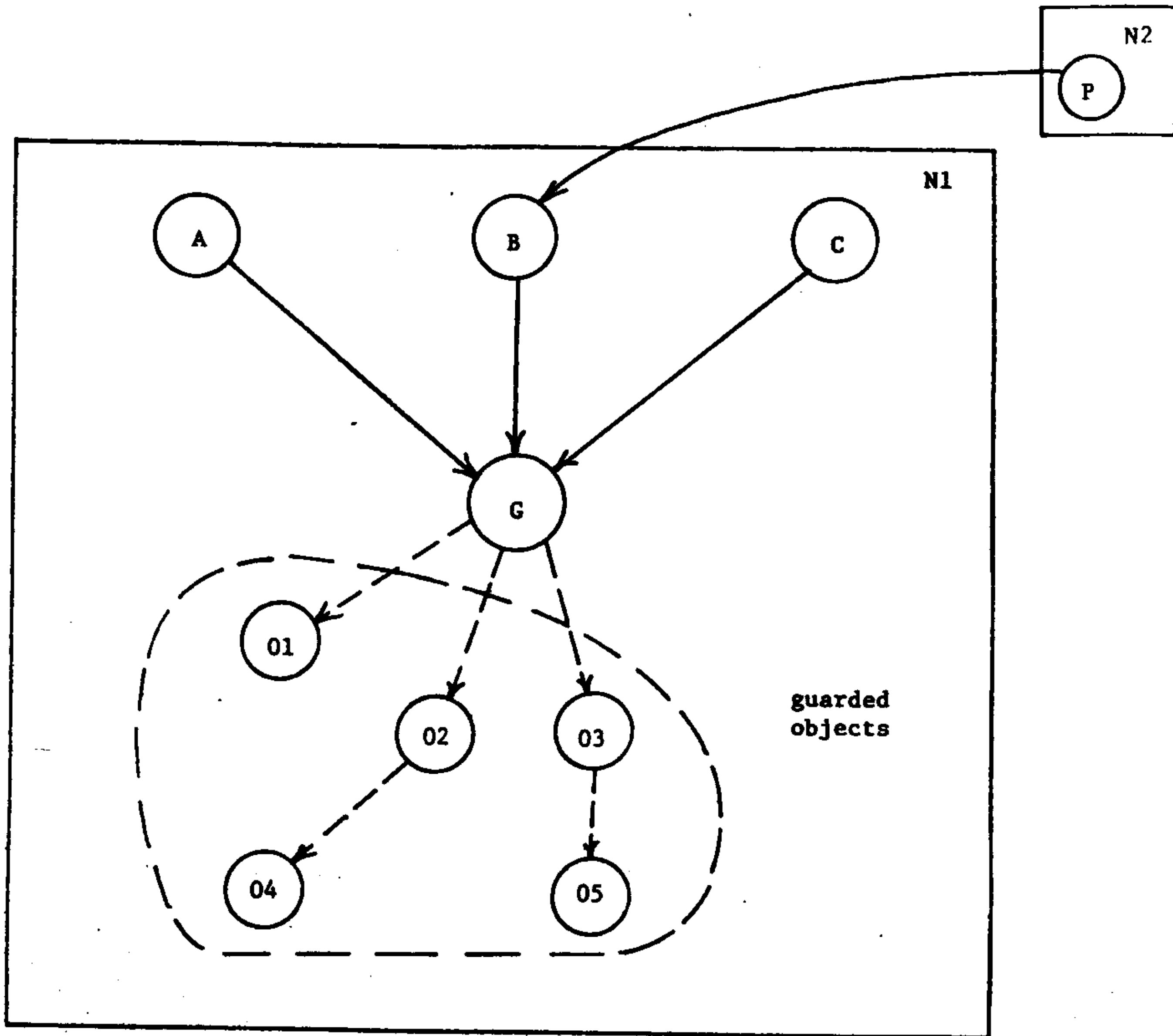
One of the basic goals of our project is to allow the application programmer to work with application-oriented entities. The same concern applies in the area of protection. That is, protection constraints should be expressible in application-oriented terms. Powerful abstraction mechanisms and the concept of abstract nodes both contribute towards this goal.

Let us look now at the inter-node protection problem in the abstract network from a slightly different viewpoint. It will be a rare case where a request occurring between nodes consists of nothing more than the reading or writing of a single primitive object. In most cases, we can expect the request to be composed of an aggregate of reads and writes on various objects, which the requesting node views as atomic. This is generally referred to as an atomic transaction. The thing that must be protected from outside is the right to execute this atomic transaction. It is quite possible that the isolated reads and writes that are required as part of this transaction are

to read certain records, others to read and write them. Each data model implies the existence of an algorithm to translate between that data model and the actually stored information. It is these algorithms that must be provided in advance, one set for each data model. The programming system must provide facilities for creating such data models, mapping them into the actual stored information, and synchronizing read and write operations originating from different data models.

We have stated this paradigm in terms of the traditional vocabulary of data management. Let us state it again in a different vocabulary, that of typed objects. An abstract type, which allows only certain well defined operations on the objects of that type, while in reality it may perform arbitrary computation on a possibly large number of objects that constitute its representation, is very close to the idea as a data model. The traditional view of data models permits a low-level information entity to be shared by different users through a variety of data models. To support this view via abstract types, it must be possible to manipulate a single low level object as part of a number of different abstract data objects, depending on the rights of the different users. The idea of data models is that different users have different views of the world, but, fundamentally, they do turn out to be views of the same world. Thus, in some sense, they must ultimately rest on the same physical data.

The inter-node protection can be enforced as follows. Any outside user (process) perceives the information in a particular (abstract) node as a number of objects that he can manipulate independently, and a set of permissible operations on those objects. These externally visible objects are arranged in such a way that there are no explicit protection constraints that tie one object to another. A message arriving at a node to manipulate one of



A, B, C: protection agents
G: guardian

Figure 3: Inter-node protection mechanism in the abstract network: the process P (abstract node N2) can reach the objects guarded by G (abstract node N1) only through the protection agent B.

object a notation describing the particular operations that this principal is permitted to perform on that object.

14. Reed, D.P., "Naming and Synchronization in a Decentralized Computer System," M.I.T. Department of Electrical Engineering and Computer Science, PhD Thesis, September 1978.

15. Roberts, L.G., Wessler, B.D., "Computer Network Development to Achieve Resource Sharing," Proc. AFIPS SJCC, (1970).

16. 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).

17. Saltzer, J.H., "Protection and the Control of Information Sharing in Multics," Communications of the ACM, Vol. 17, No. 7, (July 1974), pp. 388-402.

18. Stearns, R.E., et al., "Concurrency Control For Database Systems," Extended Abstract, IEEE Symposium on Foundations of Computer Science, (October 1976), pp. 19-32.

19. Svobodova, L., "Distributed Computer System in a Bank: Notes on the First National City Bank," M.I.T. Laboratory for Computer Science, Computer Systems Research Division, Request for Comments No. 155, (January 23, 1978).

20. Svobodova, L., "Distributed Computing in the Bank of America," M.I.T. Laboratory for Computer Science, Computer Systems Research Division, Request for Comments No. 157, (February 17, 1978).

21. Thomas, R.H., "A Solution to the Update Problem for Multiple Copy Data Bases which Use Distributed Control," Bolt Beranek & Newman, Inc., Report No. 3340, (July 1976).

22. Wirth, N., "Modula: A Language for Modular Multiprogramming," Software Practice and Experience, Vol. 7, No. 1, (January 1977).

23. Wulf, W.A., et al., "An Introduction to the Construction and Verification of Alphard Programs," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, (December 1976), pp. 253-265.