Tandem TR 85.4

AN APPROACH TO DECENTRALIZED COMPUTER SYSTEMS

Jim Gray

June 1985

Revised January 1986

ABSTRACT

The technology for distributed computing is available. However, decentralized systems still pose design and management problems. Decentralized systems will always require more careful design, planning, and management than their centralized counterparts.

This paper begins with the rational for and against decentralization. Then, a technical approach to decentralized systems is sketched. This approach contrasts with the popular concept of a distributed integrated database which transparently provides remote IO against single system image. Rather, it proposes that function be distributed as "servers" which abstract data as high-level operations on objects and communicate with "requestors" via a standard message protocol. The requestor-server approach has the advantages of modularity and performance.

This paper has been submitted for publication to the IEEE Transactions on Software Engineering.

TABLE OF CONTENTS

1. What is a decentralized system?

A decentralized computer system, as opposed to a centralized one, is a collection of autonomous computers which communicate with one another to perform a common service. A decentralized system might occupy a single room, but more typically decentralized systems have geographic and organizational diversity.

The world telephone system is the biggest and best example of a decentralized system. It consists of thousands of computers, and almost a billion terminals. Some of the nodes of the system are tiny local PBX's while others are quite large, able to handle many calls per second. Different parts of the system are operated by cooperating organizations with different hardware, different languages and different ideologies. They have agreed to protocols which allow direct dialing from anywhere to anywhere and the consequent automatic routing and billing. [Amos].

Other examples of decentralized systems can be found in the world travel industry system, inter-connecting travel agents with hotels, airlines, and other reservation systems, and the emerging electronic financial system, connecting financial institutions, businesses and governments.

These systems have the following common features:

- diverse organizations and organizational procedures,
- diverse computer architectures, both hardware and software,
- diverse terminal types,
- diverse system sizes, from tiny to large, and
- diverse site environments.

The "glue" that holds each of these systems together is a message protocol. For each of these systems, the participating organizations agreed that: "When I send you this message, it means thus and so". They also agreed to the bit-level format of each such message.

The thesis of this article is that these systems could have been built as an "integrated database", but that such a system would be a management nightmare. In reality, the parts of these systems are not tightly integrated -- each part is generally quite different from the others. It is "integrated" by having the parts agree to a common but arms-length message protocol.

But, before getting into the technical details of distributed computing, it is worthwhile to review why we distribute computer systems in the first place.

1.1. Why decentralize systems

There is no "best" form of organization. Each form has its advantages and disadvantages. The structure of computing is just one aspect of the structure of an organization. Centralized organizations will continue with centralized computing and decentralized organizations will adopt appropriate degrees of decentralized computing [March].

For such decentralized organizations, distributed computer systems are likely to give the operational units control of their data. If done correctly, a decentralized system allows more flexibility for growth of capacity and function. In addition, a decentralized system can be more maintainable since each part can change anything so long as it continues to support its external interfaces.

Expanding on this argument, there are both organizational and technical motives for decentralizing a system.

The main organizational reasons for decentralized computing are:

- Integration of -existing systems: As existing computer systems are connected to provide common services, one naturally gets a distributed computer system. The world phone system, travel reservation systems and finance systems give good examples of this decentralization via integration of pre-existing systems.
- Organizational autonomy: Managers want administrative control of facilities critical to their effectiveness. To make this control real, most administrators also want operational control of their computers and their databases. Increasingly, computer systems are being designed to reflect the organization structure.

There are several technological reasons for building a decentralized system. It is sometimes not feasible to build a centralized system with the required capacity, response-time, availability or security of a distributed system. Typical issues are:

- Capacity: Some applications have needs far in excess of the largest computer. They can't fit on a single computer or a single site.
- Response-time: If requestors are distributed over a large area, the transit times for requests and replies may be prohibitive. Long-haul communications can add a second to response time. Putting processing and the needed data close to the requestor eliminate such delays.
- Availability: Having geographically distributed autonomous sites processing parts of the same application has the effect of error containment -- a failure or

disaster is likely to be limited to a single site. In addition, the remaining sites may be able to substitute for a failed node.

- Cost: Decentralization may reduce long-haul communications traffic and hence reduce communications costs. The communications savings may outweigh the extra cost of distributed facilities and staff.
- Security: Some organizations feel more secure when they have physical control over the site that stores their data. (I am skeptical of this argument but it is frequently made).

Modularity is another reason for building a distributed system -- the desire for modular growth and change of function and capacity as well as the desire for the manageability that derives from a modular structure.

A decentralized system is necessarily modular: the autonomous nodes of the computer network communicate with one another via a message protocol. In such a system it is easy to add capacity by adding nodes or by growing a node. If done properly, such change is non-disruptive -- changing one service does not interrupt other services so long as the change is upward compatible. Centralized systems are much more limited in their ability to grow and change.

1.2 Integrated system or integrated database

As a preview of the thesis of this paper, observe that a major thrust of the 1970's was towards the centralization of data and application definition so that diverse parts of the organization could share information -- the goal was an "integrated-database". In most cases this has resulted in a bureaucracy and a monolithic system. The corporate-wide Data Base Administrator (DBA) is a centralized function and has all the problems associated with centralized systems. This structure can be a source of delays and organizational friction.

A more workable approach to decentralized systems is to let each department have its own DBA. Rather than exposing the detailed record formats of its database -- the traditional view of an integrated database -- each function externalizes a protocol for manipulating the data it maintains. For example, the order entry function of a business might support messages to lookup, add, and alter orders. The database and accounting rules for managing orders would be hidden within the procedures supporting these messages. Such an interface allows a department to change its internal implementation and to add new function without impacting other departments, so long as the old message interface is still supported. In addition, the department can enforce integrity and authority constraints on the data by assuring that only its programs alter the data. This gives modular change and modular growth to the whole system. Management of these protocol definitions becomes the responsibility of the network DBA.

2. Technical aspects of decentralized systems

So far, the rational for decentralized systems has been sketched. This section proposes a technical approach to decentralized systems.

The main technical problem unique to decentralized systems is the lack of global (centralized) knowledge. It is difficult to know everything about the rest of the network. Yet global knowledge seems to be required to answer questions such as: "Where is file A?" or "What is the best way to reach node N?". Most other technical problems of decentralized systems are also present in centralized systems. It is simply a matter of degree. Messages in a distributed system travel more slowly, less reliably and at greater cost. In a centralized system, a data item is rarely more than a disc seek away (~ 30 ms), in a distributed system it may be several satellite hops away (~1 sec). Moreover, long-haul communication systems sometimes lose messages and always charge a lot to deliver one. The lack of global knowledge adds an entirely new dimension to the problems of data placement and program execution -- where to keep the data and where to run the programs.

The following model deals with decentralization of knowledge by making each component a module which may be connected to any other module.

2.1 Computational model: objects-processes-messages

Much in the style of modern programming languages such as Smalltalk, Modula, and Ada, the system supports the concept of object "type" and object 'instance". Primitive object types are Process, Terminal, File, and Node. Application developers implement new types by writing programs in a conventional language such as Cobol or PLI. The programs are executed as processes -- programs running on computers. Externally, each process appears to be an object instance which supports the verbs of that object type. Internally, processes are structured as a collection of sub-routines which implement that type.

A process may address other objects (processes, files, terminals, etc.) by OPENing them . The open step takes a symbolic name for the object, locates the named object, checks the authorization of the process to the object and returns a reference for a "session" to the object -- in operating systems terms, it returns a capability for the object. Thereafter the process may operate on the object by sending messages via the session (WRITE), and reading the replies (READ). Ultimately the session is terminated (CLOSE).

This discussion is symmetric, a process may be OPENed. In that case it will receive an OPEN message, which it may accept or reject, and if accepted it will receive a sequence of requests which it READS and generates a corresponding sequence of replies (WRITES). Ultimately it may receive a CLOSE message or unilaterally close the session.

Object creation is accomplished by opening a session to a process managing the object type. This session is then used to communicate the name and attributes of the new object.

To give a simple example: One opens a session to a file to access the file. The session is opened with a process (file server) representing that file. Sending messages via the session instructs the server process to position within the file, and insert, update or extract records in the file.

Because every object has a name, a process can address any object in the network -- subject to authorization restrictions. This means that the process is unaware of the physical location of the object. It can write on any terminal, read or write any file, and address any other process. This is one key to structuring a decentralized system.

In order to fit into the CALL structure of conventional programming languages, the concept of "remote procedure call" is introduced. Rather than the program having to

OPEN, WRITE, READ and CLOSE the object, it may CALL an operation on the object. A remote procedure call has the format:

<operation>(<object>,<pl>,<p2>,…,<pn>)RETURNS(<r1>,…,<rm>)

where the <pi> are by-value parameters and <ri> are returned results. This request is translated as follows:

(1) If the process managing the object is not OPEN, an open message is sent to it.

(2) The following message is sent to the object manager:

(<operation>,<object>,<pl>,<p2>,…,<pn>)

(3) The object manager performs the operation and replies with message:

(<rl>,<r2>,...,<rm>).

(4) The reply message is unpacked into <r1>, …,<rm>.

Typically, the underlying software "saves" the open so that later operations on the same object will have low overhead.

This structuring allows functions to be executed within a process, or in a different process perhaps at a remote node. It gives a uniform way for a node to export its primitive types, and abstractions of these types. As will be argued later, nodes generally export high-level abstractions rather than primitive types.

The message formats (2) and (3) above describe the interface to the type. Defining a decentralized system consists of defining these message formats and their semantics. Thereafter, independent organizations can implement these protocols to provide services or to make requests of others.

This model is the basis for the Tandem system [Tandem], Argus System [Liskov] and the R* System [Lindsay]. It also forms the basis for IBM's SNA Logical Unit Type 6, which defines the types DL/l Database, Queue, Process, etc., and will grow to include many more types [Hoffman]. The airlines reservation system to manage seats, the electronic funds transfer systems also use the message model.

2.2 Dictionary: naming, authorization and control

The system dictionary plays a key role in object-oriented systems. The dictionary maps symbolic names to object descriptors, and performs authorization checks. In addition, the dictionary provides some reporting and control functions.

In order to structure the name space, symbolic names are organized as a hierarchy. The name “A.B.C” names an object but may also name a subtree of objects each with “A.B.C” as a prefix of their names. Interior nodes of the hierarchy act as directory objects. Any object can act as a directory object. Allowing names to grow on the right allows objects to have sub-objects. Allowing names to grow on the left allows addressing of other name spaces for inter-networking. Prefixes and suffixes on telephone numbers exemplify this principal.

The dictionary is partitioned among the nodes of the network; each node has a part of the dictionary. The dictionary stores the generic attributes of each object, its name, type, owner, and authorization, in the name space. The manager for that object type stores type specific attributes in catalogs which parallel the name space. In this way, a new object type can be added by creating programs/processes which maintain the catalogs of that type and a new type instance can be added by requesting one of these processes to add records to the catalogs for that type as well as making an entry in the name space.

When an object name is presented to OPEN, it is looked up in the name space. The prefix of the name designates the partition of the name space. For example Lookup A.B.C.D might look for “B.C.D” in the name space at node “A”. The alias object type allows renaming of objects or directories. Aliases are generally followed until a non-alias object or error is encountered. Lookup returns a descriptor for an object. The OPEN procedure uses the information in this descriptor to route the open message to the appropriate object.

Node autonomy implies that each node must be able to operate in isolation. Hence, each object or object fragment at a node is described by the name space at that node. In addition, the node dictionary may have descriptions of objects at other nodes. For example, if a file is partitioned among several nodes of the network, each node will certainly store the description of the local partition; but, each node may also store a replica of the description of the whole file.

Authorization is one of the most difficult areas of decentralized systems. Of course, all network traffic is encrypted and low-level protocols do node authentication. Each process executes under some authority (ID). Each object has an access control list associated with its descriptor in the dictionary. The entries of each access list are <who,what> pairs.

The who" is either an ID or the ID of a GROUP of IDs. The "what" is a list of authorities allowed to that object.

When a process tries to OPEN the object "Z", the name "Z" is looked up in the dictionary to find its descriptor and access control list. The name server first checks the access list to see if the requestor has the authority to open the object. If not, the lookup signals a security violation.

The issue of authenticating the requestor arises when the OPEN travels across the network. At best, the server knows that requestor R at node N made the request -- i.e. node N will vouch for R. Either of the following two approaches deal with this: either the access list can be structured as "ID at NODE" elements for the "who" fields or a bidirectional password scheme can be required for remote requestors.

As explained so far, the dictionary implements and exports objects of type directory, alias, type, access control list, and group. The support of types allows others to implement basic types such as record, file, terminal, and application types such as invoice and customer.

In addition to providing the basic naming, authorization and type extension facilities, the dictionary also provides a structure for reporting and control. The dictionary implements an object of type "dependency". Each dependency is a binary relation among objects. When a file definition is based on a record definition, the relationship

< FILE-A , RECORD-R >

is entered into the DEPENDS-ON relation. When an index is added to a file, the relationship

< FILE-A , INDEX-I >

is added. When a program is compiled from source S1, S2, S3,.. SN, the relationships

< O , S1> and < O , S2 >, … , < O , Sn>

are added.

Centralized systems are able to recompute relationships on demand rather then maintain them as they are created. A decentralized system cannot economically search the entire system looking for relationships. Hence, the relationships must be explicitly asserted and maintained.

Relationships can be used for reporting and for control:

- Reporting: Relationships can tell what objects will be affected by a change and in turn can be used to automatically propagate the change in rebuilding an application -- recompile all affected programs, and reorganize all affected files and displays.
- Control: Relationships can be used for change control -- to freeze them and all the objects they depend upon.

2.3 Data Management

2.3.1 Data definition

Database systems implement the "record" type and operations on records. A record definition specifies the record fields, each field storing atoms of a designated type. In addition, integrity constraints may be associated with the record definition. Record instances must satisfy these constraints. Uniqueness, value range, and referential integrity are typical integrity constraints [Date]. In addition, record definitions may have physical attributes such as location, recovery attributes, and block size. The instances of a record type are variously called a file, relation, set or table. We use the term file here.

A file may be partitioned among nodes of the network based on some predicate-to-node map. The following is a sample partitioning criterion:

ACCOUNT.NUMBER < 1000000 <=> WEST

ACCOUNT.NUMBER ≥ 1000000 <=> EAST

A record instance will be stored at the node (partition) which satisfies the predicate.

Data is partitioned for several reasons:

- Load sharing: distributing traffic on the data,
- Locality: placing it closer to the data consumers thus improving local availability and response time and,
- Autonomy: allowing local control of each partition of the data.

Partitioning has long been used to distribute data across multiple disk arms at a single node. Geographic distribution is a simple generalization of that concept. The implementation details are slightly different since the record definition must be replicated at each dictionary of each involved node.

A record may be replicated at several nodes. Replication at a node has long been used for availability. If one copy is lost, the others can be used in its stead. In a geographically distributed system, replication can have the added benefit of improved response by eliminating long-haul message delays.

But replication at multiple nodes connected via communications lines presents several novel problems if updates must be broadcast to all the replicas. This may cause substantial delays, and may inhibit update if some replicas are inaccessible. The presumption in a centralized system is that “if a replica is unavailable to me, it is unavailable to everyone”. This assumption is not valid in a decentralized system. Unavailability of a replica can be caused by replica failure or by communication failure. In the latter case, a replica may be available to one part of the network but not to others.

The concepts of current and stale information help to understand the algorithms designed to deal with the maintenance of replicas. Quite general algorithms to maintain data currency are known, but they have substantial message delay cost. Other algorithms are known which trade off data currency in exchange for better performance or availability. Three kinds of replication are worth considering:

- Current Global Replicas: In this scheme a majority of the replicas are updated as part of each transaction (the definition of majority differs from proposal to proposal). The best of these algorithms tolerate some node unavailability but they have the problem that for a small number nodes (two) they do not tolerate data unavailability and for a large number of nodes (more than 2) they introduce long delays in transaction commit [Gifford], [Abadi].

- As Soon As Possible (ASAP) Updates: This scheme postulates a master node (per record type or instance) which is allowed to unilaterally update the record at his node. The updates are then asynchronously sent to the other replicas. This approach sacrifices consistency for availability and response time [Norman]. If the system is operating correctly, replicas will be only seconds out of date.

- Snapshots: There is a master file somewhere and slave copies elsewhere. Each slave copy is a snapshot of the master as of a certain time. The slave copies are periodically updated. This is appropriate for files which change very slowly and for which currency is not critical [Adiba]. In this design, replicas may be a day or a week out of date.

Each of these approaches to replication has its place. Notice that the dictionary must use a current data algorithm for managing the replication of object descriptions because its users may not be able to tolerate inconsistencies.

The replication criterion and replication algorithms are specified when defining or redefining a record. When a program runs, it must be unaware that a record is partitioned or replicated because the partitioning and replication may change after the program is written. In order to deal with this, the program may specify that it is willing to accept stale data

```
CURRENCY := FALSE
```

or that it wants the most current copy

```
CURRENCY := TRUE
```

This approach allows data independence and also allows performance improvements which make replicated data workable.

2.3.2 Data manipulation

Record instances are accessed and manipulated by a data manipulation language (DML) which allows the requestor to add, read, alter and delete sets of records. The DML is really a compiler for a limited form of set theory expressions: It must contend with records distributed over several nodes of the network.

The DML compiler produces a plan to perform the desired operation. The compiler is aware of the data distribution criterion and based on that it picks a global plan which minimizes a cost function. It then sub-contracts the access to the local data to the remote nodes. These nodes develop their own plans for doing the data manipulation at their nodes. If the data organization at a node changes, it recompiles its plan without affecting the other nodes.

Subcontracting set-oriented operations rather than record-at-a-time operations produces substantial savings in response time and message traffic. However, even greater savings can be achieved by subcontracting the application logic itself. For example, reserving an airline seat at a remote node will involve updates to multiple record types and several integrity checks. Application level protocols cast this as a single remote procedure call to a server process that then does local DML operations. This is more efficient than invoking several DML operations across the network. It also allows the server node better control of access to the data.

2.3.3 Data independence

It must be possible to move, partition, replicate, and physically reorganize data without invalidating programs which access it. The DML plan is automatically reevaluated if the physical structure of the data changes. This is called data independence.

In addition, it must be possible to alter the logical structure of a record -- for example add fields to a record type or replace one record type with two others that can be joined together to form the original. There are limits to what can be done here, but the basic trick is to define the "old" interface as a view of the new interface. A view is a materialization rule which allows instances of one record type to be materialized from others.

Unless the view is very simple, a one-to-one map from the base records, it may not be possible to reflect updates on the view down to the base files. For this reason, views are primarily used to subset a record type to provide access control to data. They allow one to hide data from programs unless they have a "need to know".

To repeat the theme, having a procedural interface to the data (a server) allows the provider of the service much more flexibility ill redefining the data without impacting others. The view constructs of data management systems are very restrictive in the kinds of data independence they support.

2.3.4 Database design

Logical database design specifies which records will be available and the integrity constraints applied to the records. Physical database design specifies how the records are stored: file organization, indices, partitioning criteria, replication criteria.

Distribution has little effect on logical database design unless the desired design has poor performance. In that case it may have to be changed as a concession to performance.

Physical database design is intimately related to data distribution. At present, database design is a largely manual process but much of the design work could be mechanized.

The first step is to instrument the system so that one can get meaningful measurements of record statistics and activity. Given these statistics and the response time requirements of the system, the database design problem can be mechanized as follows: Since the DML can predict the cost of a particular plan for a particular database design, one can evaluate the cost of a particular database design on a particular application workload -- it is the weighted sum of the costs of the individual queries. By evaluating all possible designs, one can find acceptable or least-cost designs. This search can be mechanized.

2.4 Networking and terminal support

2.4.1 Network protocols

The discussion so far presumes a very powerful networking capability. It assumes that any process in any node can open a session with any process in any other node. Files and terminals are viewed as the processes which control them. Implicitly, it assumes that these sessions may be established with relatively little overhead.

This view is implicit in IBM'S System Network Architecture [Hoffman], Tandem's Guardian-Expand architecture [Tandem] and the ISO reference model for Open Systems Interconnection. These protocols go on to architect higher-level objects such as processes, queues and files based on the session layer. In addition they specify many layers below the session layer.

The point of this discussion is that it is reasonable to assume a very powerful networking capability in which any process in any node can open a session with any process in any other node. The only debatable point is the cost of such sessions. With present systems, the cost is high, but it seems to be acceptable given the benefits which accrue. Several commercial systems operate in this way today [CICS], [Tandem].

2.4.2 Terminal support -- terminal independence

The real revolution in data management has been in the area of data display, not data storage.

Information storage has not changed much in the last twenty years. We had random access memories, discs and tapes then and we have them now. The cost per byte has declined dramatically, but the logical interface to storage has not changed much.

In the same time, we have gone from card-readers to teletypes to alpha-numeric displays. We are about to go to bitmap displays with the heavy use of icons and graphics.

When discs get bigger or faster, the files grow and the programs continue to work. When terminals change from card-readers to bitmap displays, the programs need to be overhauled. This overhaul is expensive. Thus one frequently sees a desk with three or four terminals on it, one for each application.

This issue has little to do with decentralized systems except that decentralized systems imply a great diversity of terminal types. It is essential that new terminals work with old programs and that new programs can be written without knowing the details of each terminal type in the network.

The solution to this problem is to provide a terminal-type independent interface -- a virtual terminal. Programs read and write records from a virtual terminal and the terminal handler formats these records into screen images for that terminal type. Four pieces of information are needed to do this:

- A detailed description of the terminal characteristics (e.g. terminal type and options).
- An abstract description of the desired screen layout (e.g. field 3 of the record should be centered at the top of the screen).
- An abstract description of the desired record layout.
- A particular data record.

Given a data record, the display manager can produce a screen image. If the record is empty, an empty template is displayed. When the user fills in the template, the display manager uses the inputs to create a record that the program can read.

Providing a record interface to terminals compares to providing a record interface to discs, it fits in well with the data structuring capabilities of most languages. This interface constitutes the bulk of the code in commercial data management systems -- the database code which deals with disks is relatively small. Terminal management continues to grow with the proliferation of terminal types and features.

2.4.3 Network management

As described here, the system consists of autonomous nodes, each communicating with the others via a standard protocol. Each node manages itself and perhaps the terminals directly attached to it. This leaves open the question of who manages the network connecting the nodes and those terminals not dedicated to any particular node.

Each node is a cost and profit center and so can manage itself. But the network is a corporate resource that carries traffic between organizations. The management of this resource, both capacity planning and day to day operations, is best done by a central authority with a global view.

The network system administrator configures the network topology and capacity. A centralized network control center monitors the network and coordinates problem diagnosis and repair. For very large networks, networks that grow together, or networks that cross cost-accounting boundaries, the network is broken into domains which are managed independently. The global function manages the interfaces among these domains.

2.5 Transaction management

2.5.1 The transaction concept

A data management system provides the primitive objects and primitive actions that read and write records, and terminals. A particular application maps abstractions such as customers, accounts and tellers into the system by representing them as records. Application procedures are mapped into transformation programs that invoke sequences of primitive actions. For example, transferring funds might read the customer, account and teller record, write the account and teller record, write a memorandum record and send several messages to the terminal.

The common business term for such a transformation is transaction. The key properties of transactions are:

- Consistency: the transaction is a consistent transformation of the abstract state (e.g. money is not created or destroyed).
- Atomicity: either all the actions of the transaction occur or the transaction is nullified.
- Durability: once a transaction completes (commits), its effects cannot be nullified without running a compensating transaction.

A general and clear notion of transaction is essential to the structuring of decentralized systems. Most systems do not have a general notion, some insist that a single database action is a transaction; others believe that sending a message delimits a transaction. Such assumptions are a liability when operating in a decentralized environment.

A simple and apparently general model postulates that a program may issue:

```
BEGIN-TRANSACTION    returns (TRANS-ID)
```

which returns a transaction identifier. Thereafter, all work done for this transaction (by this process or others) is tagged by this transaction identifier. The program may now spawn processes, make local and remote procedure calls and generally work on the transaction in any way it likes. Any participant of the transaction may issue the verb:

ABORT-TRANSACTION (TRANS-ID)

which will invalidate the transaction identifier and will nullify (undo) all actions of the transaction. Alternatively, any participant may issue the verb:

COMMIT-TRANSACTION (TRANS-ID)

which will cause all the operations of the transaction to be made public and durable. The commit operation must query all participants in the transaction to assure that they are prepared to commit. If this query fails, the transaction is aborted. Prior to commit, the system may unilaterally abort a transaction in order to handle overloads, deadlocks, system failures and network failures.

This model is implemented by the Tandem system [Borr] and, except for the absence of a network unique transaction identifier, is specified by SNA and implemented by CICS [CICS].

A slightly more general model in which transactions may be nested within one another is desirable. As it becomes better understood, this idea will probably be considered essential [Gray], [Liskov].

Whatever design is adopted, all the data management systems in the network must agree to the same transaction commit/abort protocols. IBM's LU6.2 commit protocols are becoming the de facto standard supported by most vendors.

2.5.2 Direct and queued transaction processing

The execution of a transaction is caused by the arrival of a message, or by an event. If the transaction is started by a message, the input message and its source are parameters to the transaction. The transaction may then converse with the source, access the database and then commit. This approach is called "direct" transaction processing. It is appropriate for transactions that can be processed in less than a minute.

Another approach is to put arriving input messages and pending output messages into message queues. This has the virtue of decoupling the submission of the transaction from its processing and from the delivery of the response. In queued transaction processing systems, processing a request consists of three transactions: (1) receive and acknowledge input message, (2) process message, and (3) deliver response.

Step (2) of processing a queued transaction cannot converse with the terminal; it must get all its input from step (1) and deliver all its output in step (3). This is the major limitation of queued transaction processing.

Queued transaction processing is ideal for applications in which there are long delays in processing the transaction (e.g. delivery of electronic mail or the request for a voluminous report). The sender does not want to wait while the transaction is processed. The ASAP updates mentioned for replicated data are an excellent example of this.

Another application for queued transaction processing arises when the transaction crosses organizational boundaries. Implementation of transaction atomicity allows a program to leave data at other nodes in an uncommitted state for an indefinite period. If one organization will not accept this exposure, it can refuse to participate in the commit protocols. In that case, it can offer queued transaction processing. For example, the airlines internally use a direct transaction processing for airlines reservation, but use queued transactions when talking to other airlines systems. Banks make a similar distinction. IBM's IMS provides only queued transaction processing, while IBM's CICS and Tandem's Encompass default to direct transaction processing with the option for an application to queue the request for later processing.

2. <u>An approach to designing and managing decentralized systems</u>

The first section discussed the rational for decentralization and touched some problems associated with decentralized systems.

The second section sketched the technical features of a decentralized system. It tried to substantiate the claim that solutions to most technical problems associated with decentralization are known and are available.

Broad experience with the design and operation of decentralized systems is lacking. There are comparatively few examples that one can look at for guidance. It takes a long time to evolve a decentralized system.

Generalizing from the few examples of decentralized computer systems in existence and from the many examples of decentralized organizations, one sees the following pattern [Anderton].

Certain aspects of the organization are common to all and so must be designed and controlled by a central organization. In computing, the industry-wide networks such as Swift, Fedwire, and Airinc are examples of this. A central organization controls this resource and the protocols for its use.

But this is just one level of architecture. Each individual bank or airline has its own internal network which is a corporate-wide resource controlled by a corporate organization. This corporate organization manages the corporate network and the protocols for its use.

Each organization within the corporation provides services to the rest of the corporation and in turn needs services from other parts of the corporation. The syntax and semantics of these requests and replies constitute the global architecture. Each individual organization publishes the requests that it honors as a network-wide service.

This refinement continues. Each level externalizes its services as a set of inputs (request messages) and outputs (responses). The corresponding system structure parallels the organization structure. Functional units supply their services to one another by processing and responding with electronic messages rather than paper or voice messages.

This message protocol is an ideal way of structuring interaction between nodes of a computer network and between organizations. It also applies to structuring communication within an application. It has the benefits of:

- Encapsulation: Encapsulating the rules for manipulating the objects (the procedures embody the rules) and hence assuring that the rules are followed.
- Abstraction: Providing a much simpler conceptual interface to the next level of abstraction. Higher layers need not know the internal rules that govern the manipulation of the accounts.
- Autonomy: Allowing change of internal procedures without perturbing the consumers of the abstraction.

The protocols are implemented as follows. When it comes from the factory, a computer system provides interfaces to records in the database and I/O terminals (in the network). These notions are too primitive to be dealt with directly and so are abstracted by application programs which provide simpler and more convenient abstractions and ways to manipulate them. For example, in inventory control, abstract objects representing parts, bins, orders and invoices, and operations to manipulate these objects, are developed. These abstractions are built, one atop the others, to arrive at services that can be externalized.

The internal and external structuring mechanisms must be the same so that the structuring can be hierarchical. Hence, a standard notion of modularity is the basis for the architecture of decentralized systems. The two key ideas are the notions of abstract data type and the transaction concept. Type operations must be defined in a way that is independent of location -- they must be defined to allow remote procedure calls. Transactions allow multiple operations to be grouped together as a single all-or-nothing operation.

Operations are defined by the message formats for requests and replies. In addition a request carries additional attributes such as:

- Security attributes of the requestor.
- Is the message a whole transaction or part of a larger transaction?
- Is the message to be processed immediately, or should it be queued for later processing?

To give a specific example, many accounting methods require double-entry bookkeeping and certain checks about when funds are credited and when they may be debited. These rules can be hidden by the procedures that externalize the verbs:

CREATE ACCOUNT
DEBIT-OR-CREDIT ACCOUNT
READ ACCOUNT
DELETE ACCOUNT
HOLD ACCOUNT
etc.

This interface is simpler to manipulate and less likely to change than are the rules and underlying representation associated with accounts.

A particular request to do a credit check on a particular customer account would have the format:

REQUEST:

FUNCTION: account management
OPERATION: credit check
VERSION: version of this message
REPLY TO: requestor id
REFERENCE NUMBER: integer
TRANSACTION-ID: transaction requesting the operation
CUSTOMER ID: customer identity
ACCOUNT ID: account being checked
SECURITY CODE: requestor's security ID

REPLY:

FUNCTION: credit
OPERATION: credit check reply
VERSION: version of this message
REFERENCE NUMBER: integer
CREDIT LIMIT: dollar amount

The first field specifies the service (organization) that handles the message, the second field specifies the operation, the third field specifies the version of the interface (later versions may have additional or changed parameters), and the remaining fields are parameters to the operation. Careful specification of the meanings of each of these fields and the meaning of the operation is part of the message definition and would be administered by the network administrator.

Presumably, the accounting department services a repertoire of requests. Its function is entirely specified by the requests it accepts and the responses it produces.

Summarizing then, the design and administration of a decentralized system proceeds as follows:

- The system is hierarchically decomposed into functions which parallel the organization information flow.
- Each function is defined by the services it provides: inputs, outputs and semantics.
- These interfaces are documented and external interfaces are registered with the system administrator.
- System architecture consists of the careful definition of these messages.
- System administration, as opposed to database administration, consists of recording and managing these message definitions.
- The function of database administration is an internal function of the individual service organizations.

A benefit of this approach is that it obviates the need for directly integrating the heterogeneous databases of an organization. Each department can use its favorite brand of computer and favorite brand of data management system as long as they all support the common transact and virtual terminal protocols.

3. Summary

I have tried to make the point that many of the reasons for decentralization run counter to the concept of an integrated distributed database.

Security, integrity, auditability, performance, changeability all are adversely impacted by having a centralized Data Base Administration controlling the entire system.

A decentralized system viewed as a federation of nodes, each autonomous of the others, is more likely to accept a requestor-server approach than a remote-IO approach. Fortunately, requestor-server designs have many technical advantages in addition to their organizational advantages. A requestor-server design sends fewer long-haul messages and they are easier to audit. The design is necessarily modular and offers more opportunities for non-disruptive change and growth.

It is no accident that all the large distributed systems we see in operation operate as requestors and servers communicating via messages.

It seems likely that the concept of an integrated database will be restricted to single computers, or at most to local networks of computers, which are managed by a single authority. Truly decentralized system will use the requestor-server approach.

4. <u>Acknowledgments</u>

The computational model described in section 2 derives largely from the Argus, Tandem, and R* system architectures. Section 2.2 outlines a dictionary design due to Dave Barton, Bob Jones, Gary Kelley, Don Maier, Mark Moore and myself.

5. References

[Abadi] Abadi, A.L., Skeen, D., Christian, F., "An efficient, fault-tolerant protocol for replicated data management", ACM PODS March 1985.

[Adiba] Adiba, M., Lindsay, B.G., "Database Snapshots", Proc Proc. 6th Int. Conf. Very Large Databases, IEEE N.Y., Oct 1980, pp. 86-91.

[Amos] Amos, E. J. "Electronic switching: Digital central office systems of the world", IEEE, N.Y., 1982, pp. 1-268.

[Anderton] Anderton, M., Gray, J. "Distributed computer systems, four case studies", Tandem Technical Report TR 85.5, June 1985, Cupertino, CA.

[Borr] Borr, A., "Transaction Monitoring in Encompass: Reliable Distributed Transaction Processing", Proc. 7th Int. Conf. Very Large Databases, IEEE NY, Sept 1982, pp. 155-165.

[CICS] "CICS/VS System Application Design Guide", Ver. 1.4, Chap. 13, IBM Form No. SC33-0068-1, June 1978, Armonk NY, pp. 379-412.

"CICS/VS Introduction to Program Logic", Ver. 1.4, Chap. 1.6, IBM Form No. SC33-0067-1, June 1978, Armonk NY, pp. 83-110.

[Date] Date C.J., An introduction to database systems, Vol. 1-2, Addison Wesley, Sept 1981.

[Gifford] Gifford, D. K., "Weighted voting for replicated data", ACM Operating Systems Review, 13.5, Dec 1979, pp. 150-162

[Gray] Gray, J.N., et. al. "The recovery manager of a data management system", ACM Comp. Surveys, 13.2, June 1981, pp. 223-242.

[Lindsay] Lindsay, B.G., et. al., "Computation and communication in R*: a distributed database manager", ACM TOCS, Feb 1984, pp. 24-38.

[Liskov] Liskov, B., Scheifler, B., "Guardians: a methodology for robust, distributed programs", ACM TOPLAS, Vol. 5, No. 3, July 1983, pp. 381-405.

[March] March, J.C., Simon H.A., Organizations, Wiley, 1958, NY. NY.

[Norman], Norman, A.D., Anderton M., "Empact, a distributed database application", Proceedings of AFIPS National Computer Conference, Vol. 52, pp. 203-217, 1983.

[Hoffman] Hoffman, G., "SNA Architecture reference summary", IBM Form No. G320-6024, Dec 1978, Armonk NY, pp. 218-248.

[Tandem] "Introduction to Tandem computer systems", Tandem Part No. 82503, March 1985, Cupertino, CA.

"Expand(tm) reference manual" Tandem Part No. 82370, March 1985, Cupertino, CA.