Report on the
Workshop on Fundamental Issues in Distributed Computing

Fallbrook, California
December 15-17, 1980

Sponsored by the Association for Computing Machinery, SIGOPS and SIGPLAN

Program Chairperson:	Barbara Liskov, MIT Laboratory for Computer Science
Program Committee:	Jim Gray, Tandem Computers Anita Jones, Carnegie Mellon University Butler Lampson, Xerox PARC Jerry Popek, University of California at Los Angeles David Reed, MIT Laboratory for Computer Science
Local Arrangements:	Bruce Walker, University of California at Los Angeles
Transcription and Editing:	Maurice Herlihy, MIT Laboratory for Computer Science Gerald Leitner, University of California at Los Angeles Karen Sollins, MIT Laboratory for Computer Science

1. Introduction

Barbara Liskov

A workshop on fundamental issues in distributed computing was held at the Pala Mesa Resort, Fallbrook, California, on December 15-17, 1980. The purpose of the meeting was to offer researchers in the areas of systems and programming languages a forum for exchange of ideas about the rapidly expanding research area of distributed computing.

Our goal was to try and enhance understanding of fundamental issues, but from a practical perspective. We were not looking for a theory of distributed computing, but rather for an understanding of important concepts arising from practical work.

Accordingly, the program committee organized the workshop around the set of concepts that we felt were significant. The sessions were organized as follows:

Session Title	Leader
Real Systems	Jim Gray
Atomicity	David Reed
Protection	Jerry Popek
Applications	Rich Rashid
Naming	Jerry Popek
Communications	Butler Lampson
What do We Need from Theory	Sue Owicki
What are Important Practical Problems	Barbara Liskov

The sessions were oriented toward discussion rather than formal presentation. Typically, in each session a few participants gave short presentations on research related to the session topic. The remainder of the session consisted of general discussion.

The sessions were recorded by three participants, Maurice Herlihy, Gerald Leitner, and Karen Sollins. The transcriptions were used by the session leaders in preparing the final reports.

The following sections contain these reports. Also included is a paper by John Shoch based on a presentation he made during the final session. A list of attendees is included as an Appendix.

2. Real Systems

Chaired by Jim Gray

This session presented several distributed systems which are in use today. The purpose of the session was to review current practices and get a statement of outstanding problems. The speakers were asked to stick to the following outline so that comparisons could be made among the systems.

What does the system do?

Does it support distributed:
- location transparency?
- replica transparency?
- failure transparency?
- concurrency transparency (consistency)?

What is the cost of:
- null operation (instruction = 1)?
- procedure call (local/remote)?

message send/receive?
transaction?

How big is the system?

What has been learned?

2.1 Jim Gray: IBM's Customer Information Control System (CICS)

Jim Gray of Tandem (formerly of IBM) described the Inter-System Communication feature of IBM's Customer Information Control System (acronym CICS ISC).

CICS is a transaction processing system, a "shell" which provides transaction services to resource managers such as DL/1, System 2000, and System R. These services include system startup and shutdown, process creation, scheduling and dispatching, error handling, device support, and transaction commit-abort control. CICS also provides a record interface to terminals and to sessions (virtual circuits). New resource managers may be added to CICS (as the database managers System 2000 and System R were added). A CICS call has the form:

OPERATION OBJECT PARAMETER1, PARAMETER2, ...

CICS provides transparency as follows: The object name may be fully qualified as SYSTEM.OBJECT or it may be looked up in a directory and resolved to SYSTEM.OBJECT. If SYSTEM is this system, the call is local and is executed. Otherwise the call is non-local and CICS

(1) packages the operation and parameter list as a message,

(2) sends the message to the destination system (using IBM's System Network Architecture protocols).

(3) The destination system creates a "mirror" process to do the transaction's work at that node.

(4) The mirror unpacks the message, and re-issues the operation. This may cause another hop but usually the object is local and the operation is executed locally.

(5) The mirror packages the response and sends it to the caller.

(6) The invoking CICS system unpacks the message and returns it as the result of the operation.

This classic remote procedure call works for queues, processes, files, DL/1 databases, System 2000 databases, and certain other object types. CICS maintains a per-system transaction log and uses the two phase commit protocol to coordinate transaction commit and deal with node and line failures.

CICS ISC has some limitations: an object and operation must reside and execute at only one node. The directories to resolve object names are made at node sysgen. CICS does not provide additional facilities for replica transparency. CICS provides the transaction notion to deal with failure transparency. The application programmer issues a SYNC (commit) verb at points of consistency. This causes CICS to commit all work of the transaction at all nodes. CICS resource managers such as IMS are responsible for concurrency transparency. They typically use locking. Global deadlocks are detected by timeout.

CICS implements IBM's System Network Architecture and is capable of talking with non-CICS and non-IBM systems. The system is about a million lines of code (mostly device support code). It can run in 300 Kbytes. The ISC code was about 15,000 additional lines (about as much as adding a new terminal type supporting the color option of the 3270 terminal). The ISC feature has been available for about 3 years. It demonstrates that remote procedure calls work, that two phase commit works, that most networks consist of one node and that users have simple needs (most use ISC for transaction routing).

Discussion:

Q: Is it inherent that most networks consist of only one node?
A: Most: yes, All: no. There are many applications for which multiple geographically distributed nodes are appropriate: airlines, EFT, and anything that has strong geographic locality of reference.

Q: Do we need transactions, or is point-to-point communication sufficient?
A: If the application does multiple updates or messages in a central or distributed system then the transaction notion eases application programming by providing failure transparency.

2.2 Jerry Held: Tandem's Encompass

Jerry Held of Tandem presented an overview of Tandem's Encompass system. Tandem emphasizes availability, on-line transaction processing, and networking. Encompass is used for on-line transaction processing applications where requests are entered from terminals and are processed by user-written application programs which access the database.

The NonStop (TM) architecture of a Tandem node consists of up to 16 processors (no shared memory) which have dual paths to all devices. In addition, devices may be duplexed. The operating system, network interface and file interface each hide single failures through the novel NonStop software architecture which duplexes paths and processes.

Encompass has the following components:

Database access methods: B-tree or sequential
Transaction manager: log plus two phase commit
Report writer plus query
Data and screen (terminal) definition
Requestor-server work flow control
(others)

Applications are typically written in Cobol and Tandem's Screen Cobol. Encompass does terminal management, schedules requests to servers running the application programs and provides database and commit services. In normal operation the processors execute different parts of the application program and also act as back-up for one another. The system tolerates single failures but a double failure will stop the system.

In practice NonStop systems have mean times to failure of about ten years (much more than one and less than 100). Operator error is a major source of failures. The system is multi-processor and message based and so generalizes well to a geographically distributed system. The network does dynamic path selection and is also NonStop. Files may be partitioned among nodes of the network based on key values of records. Files may be replicated on different disks on one system, but geographic replication is not directly supported. File names have a home location built into them and so the application is aware of the location at which the file was defined. Encompass provides transactions and record locking. Deadlock detection is via timeout. Encompass is several hundred thousand lines of code and represents about 25 man-years work.

We have learned from this design that it is advisable to take small steps at a time. There is a real need for multi-node transactions in many application areas. There are not many people using distributed transactions right now, but a lot of people are interested in using them in the future.

2.3 Roger Needham: CMDS

The Cambridge Model Distributed System consists of a collection of processors joined by a ring net. CMDS provides the following services: resource management, name lookup, printing, and authentication. Users of CMDS are allocated individual machines by the resource manager. The resource manager also allocates an "ancilla" to mediate between the user and the machine, to abstract away from the details of the particular kind of machine allocated.

A major problem is maintaining the replicated and distributed data base. When a user asks for information, it is necessary to find the appropriate registry server. Crashes and failures should be hidden (as much as possible). Locating the instance of a server that is "closest" to a client is also a problem.

Grapevine includes about 1200 computers, 1200 users, and handles about 10,000 messages per week. There are two Grapevine servers, consisting of 28,000 lines of Mesa. In conclusion, the distributed naming data base was both harder and more interesting than expected.

3. Atomicity, Reliability, and Synchronization

Chaired by David Reed

This section was devoted to a discussion of atomicity, reliability, and synchronization mechanisms and their relationship to distributed systems. Most of the discussion centered around the questions of how important such mechanisms are in distributed systems, and what role such mechanisms can/should play in design of distributed applications. Little discussion was devoted to actual synchronization mechanisms or recovery techniques, though the participants included many who have worked on implementation issues.

Perhaps the most significant results surrounded the issue of the utility of atomic actions as a built-in system feature. There were three primary points of view, as follows:

> Atomic actions are a powerful tool. They enhance modularity of programs by allowing module behavior to be defined without reference to how the implementation responds to concurrent access or failure (David Reed).
>
> Atomic actions are only one way to preserve an invariant. With a better understanding of the problem, better solutions are possible (Leslie Lamport).
>
> Atomic actions at a single site are valuable. Multi-site atomic actions may not be worth the cost or the loss of site autonomy (David Clark).

At first glance these positions seemed quite different, but upon reflection, they seem more compatible.

The following request was made of all speakers, as preparation for the session: Please prepare a 10-minute talk explaining your views on how these concepts fit into distributed systems. The following questions may be used as a guide.

> The concept of atomic actions provides a particularly simple way to organize recovery and synchronization requirements. How general is this concept? What kinds of recovery and synchronization are difficult to organize using atomic actions? Do these concepts apply outside of transaction systems?
>
> Are there other concepts that help in organizing recovery and synchronization for applications? Are there weaker forms of synchronization that would suit many applications?
>
> Can and should failures be hidden from the application?
>
> What is the cost of mechanisms to ensure atomicity? Should use of these mechanisms be applied universally?
>
> What are the costs and benefits of replication?
>
> Is it possible to make meaningful and useful tradeoffs among reliability, availability, throughput, and response time by varying parameters such as degree of replication, choice of protocol, etc.?

3.1 David Reed

An atomic action is defined as follows:

a computation that behaves as if it executes to completion or not at all. (failure atomicity)

all of its accesses to shared data either precede or follow any access outside the action. (synchronization atomicity)

From this we can conclude that states internal to an atomic action are not observable outside the action. It is also true that such atomic actions have effects equivalent to serial actions.

Modularity is supported by atomic actions. Failure atomicity helps in two ways. First, errors can be reported in terms that are understandable to the atomic action's client. Second, implementation invariants are preserved automatically. In order to make these benefits available to nested modules, composability of atomic actions is needed. Modularity is also aided by atomic actions, since synchronization requirements can be specified without requiring help from outside the module. A fully general system should support a data-dependent locus of control, rather than requiring pre-declaration of the data to be accessed by an atomic action.

3.2 Charles Davies

Charles Davies briefly addressed a number of aspects of atomic actions. First, transaction semantics imply that such an action is logically a single unit, although it may be physically interruptable. Second, there is currently one type of commit, but in reality there are layers of commits. The commit boundary may be longer than the transaction boundary. Third, given atomic actions, recovery from any action is available, although we must consider the cost of being able to try an action again, possibly in a different way. Fourth, parallel actions make logging and backing out of some actions difficult. The auditing provided by atomic actions helps solve these problems.

In addition, Davies made a number of points about atomic actions. A sampling of these are: (1) that failures should be hidden from the applications when this is agreed upon; (2) a mechanism for atomicity should be applied universally despite the cost; and (3) such a mechanism may provide replication at no extra cost, if copies are already kept in order to achieve atomicity.

3.3 David Gifford

David Gifford agreed with David Reed's definition of atomicity and put it this way. Atomic actions have the properties that they are all or nothing and that they provide serial consistency. The problems with a system that implements atomic actions occur when it fails, and in reality it will sometime. We can only reduce the probability to an arbitrarily small amount.

A system should provide atomic actions for three reasons: (1) they are required for some applications; (2) they are less expensive (Gifford estimates 1.1 to 2.0 times less expensive) than at the application level; (3) it is reasonable that a shared file system should have them. The costs of replication lie in delay and resource usage for storage and communication. The benefits lie in enhanced availability of data, reliability if a copy is lost, capacity in allowing parallel accessing, and less delay in locating nearby copies of the data. Replication can affect some tradeoffs. Based on the frequency of reads to writes, replication parameters can be used to adjust performance. Availability and reliability can be enhanced by more replication. Finally, within certain constraints delay and capacity can be improved by replication.

Conclusion: Atomic actions make it easy. Replication is good. And you need atomic actions in many cases.

3.4 Jerry Popek

Jerry Popek's discussion dealt with the following question: If a data object is replicated and the network is partitioned can one continue working on both copies (especially if transactions are available)?

Consider working in a UNIX type file system where each directory can be contained in only one other, but a file can be contained in more than one directory. If a partition occurs, then directory and file modifications on replicated directories and files could lead to inconsistency after merging.

What happens when the network merges after a partition? If only one copy has been modified, there is no problem. In the case where there have been parallel modifications, there are many interesting applications that permit automated reconciliation of version conflicts, for example directories and mailboxes in the system Popek's group has built. In these cases additions and deletions can be merged. Certainly in the cases of electronic funds transfers and airline reservation systems, these problems have been solved. If two files of the same name are created in different partitions, one may have to go back to the users to resolve the problem.

Popek's group at UCLA has built a version of UNIX that has a mechanism for automatically merging compatible changes to replicated objects when partitions merge. Popek believes that his approach is an interesting alternative way to deal with network failures.

Concerning the issue of horizontal vs. vertical partitioning of function in a distributed system, Popek argued for vertical layering because crossing the boundaries is easier (i.e. he proposed that all machines be capable of any function). This allows for graceful degradation. Horizontal partitioning makes the protocols easier, while vertical partitioning provides potentially better performance.

3.5 Dave Clark - Atomicity is Bad (for the sake of an argument)

Dave Clark proposed having mechanisms for atomicity at a single site, and no built-in mechanism for building an atomic action that acts at several sites.

Distributed atomic updates may be solving a non-existent problem. For example, in banking a customer wants to transfer some money between two accounts. He goes to a teller who logs the transaction before executing the credit and debit. The teller must be able to go back to the audit log and figure out which transactions have been completed and which have not. The two sites must be able to recognize duplicate requests and return the same responses. Two issues are important here: (1) transactions need unique identifiers, so that duplicate requests can be recognized, and (2) the audit log must be stored in stable storage so that it will not be lost. Instead of a general atomic action mechanism, an "undo" operation should be provided that undoes a transaction at a site.

Systems people take away functionality from applications programmers until the system becomes too cumbersome. In this case, the function should be left to the application.

Reed: The only requirements here are audit logs and local atomic actions, but this is all that is needed to implement atomic actions.

Clark: But in this case the local machine never gives up control, whereas in atomic actions it does at least for a moment.

3.6 Discussion

The topics discussed were distributed atomic actions (following up Clark's argument) and replication performance. Some of the comments made appear below.

Distributed Atomic Actions

Lampson and Lindsay: Save points give you only failure atomicity; they provide Clark's "undo." Intermediate states due to failures and parallelism may be visible and acceptable if they are expected.

Lamport: Atomic actions are a crutch. They allow one to serialize if one doesn't understand what is going on. One can certainly create erroneous transactions, ones that are unreasonable. Up to a certain level one may have atomicity, but not above that.

Lamport: What is the consistency that we are trying to achieve? It is not really serializability, but some invariant. **(Schneider, also)** Maybe we don't really need locking, etc. to achieve what is desired.

Reed: The users really view actions such as withdrawing money from a bank as atomic, as opposed to what Dave Clark said.

Davies: What Dave Clark was really talking about was pre-commit.

Clark: Backout (undo) must always be available, although it may get harder with time.

Lindsay: Programmers like multi-site atomic actions, but Dave Clark argues for a kind of non-atomic transactions. He hasn't shown any reason to not do the other.

Gray: Consider an example of an insurance transaction, where a person has an accident and puts in a claim. In a transaction system, the person cannot pay a premium on that insurance policy until the claim has been settled, even if the case is not settled for six months.

Schroeder: Jim Gray is confusing an insurance "transaction" and a computer "transaction". An insurance transaction is not what we have called an atomic action. What Gray describes might better be called a "saga".

Fischer: We are focussing on only undo (backward recovery). What you really want is to get to some good state, even if you weren't there before. If a few files are destroyed, you don't want to put all the files back to their state a week ago.

Replication performance

Gifford: Gifford described the replication techniques described in his paper in the Seventh Symposium on Operating Systems Principles and then concluded:

> To do replication on fast and slow machines put weak replication on slow machines.

Popek: Maybe we shouldn't use atomic actions to enforce consistency among replicated objects.

Gifford: When you don't know much about the object semantics, atomic transactions may be helpful.

4. Protection

Chaired by Jerry Popek

To help set the context of this session, the speakers were asked to explain how their work addressed each of the following issues:

> policy - the user visible rules under which protection decisions are made.
>
> low-level mechanism - how is the policy enforced?
>
> system structure - what portions of the system are involved in correct operation of the protection system; i.e., how much must be trusted?

authentication - how are the end users (or internal components) identified to one another?

In addition, it was noted that:

Encryption is one essential ingredient of an enforcement mechanism. (Unlike in centralized systems, we do not have confidence in the hardware, i.e., the wires, in this case.)

System structure strongly affects overall safety.

We shouldn't waste time talking about the strength of DES, the issue of public vs. private key based algorithms, or the role of NSA.

4.1 David Reed

David Reed described the use of encryption in a system consisting of a data repository and different brokers using the data objects. This distinction is made so that the repository need not provide a guarantee against unauthorized disclosure, but only a guarantee not to lose data. The repository keeps all versions.

In this approach each object has an associated name (UID) and key (OBK). The key corresponds to a capability which is only known to the broker. To provide protection against disclosure, the value of the object will be in clear text only when it is in use by the broker. Thus, when the broker sends an object to the repository, it sends

(o, pt, cr, {value}OBK)

where {A}B means A encrypted with key B, o is the object, pt is the pseudo-time, and cr is commit information needed by the repository. To retrieve the object, the repository sends to the broker:

(o, pt, {value}OBK).

As an enhancement to provide protection against undetected modification, we require that, to store an object, the broker send to the repository:

(o, pt, cr, {value, ck}OBK)

where "ck" is a checking field that depends on o, pt and value. To retrieve data, the repository sends to the broker:

(o, pt, {value, ck}OBK).

Then no one can successfully forge a version without knowing OBK, since the broker can detect attempts.

To protect the repository from being flooded with forgeries, a "write authentication key (WAK)" can be associated with each object. In this case, the broker sends an object to the repository by sending:

(o, pt, cr, {value, ck}OBK, wc)

where wc (write checksum) is the result of applying a function f that depends on WAK to (o, pt, cr, {value, ck}OBK). Note that exposing WAK does not allow unauthorized modification.

The name server is capable of mapping between the text name of a service, process, or computer, and its associated ring net address. Authentication is done using the name server. The file server works on a remote procedure call basis, supporting transactions on single files.

Some observations:

Communication code is expensive, with the cost dominated by checksums and copying. Doing checksums in hardware might help, but may be rather inflexible.

Accountability is expensive. As much as possible authentication should be performed locally. Remote procedure call is slow enough without having to authenticate each request.

To what degree should the name server, authentication server, or file server be centralized? Some activities benefit from an economy of scale derived by locating a single function on a single machine: e.g. a large disk used by a file server. Authentication seems to require a logical (as opposed to physical) centralization.

In summary, it is felt that the problems presented by this distributed system were essentially the same as those encountered in centralized systems; however, the solutions are harder.

2.4 David Russell: Distributed UNIX

Distributed UNIX is an experimental system consisting of a collection of machines running a modified version of UNIX.

Communication among processors is performed by a packet switching network built on Datakit hardware. The network has a capacity of roughly 7.5 megabits, shared among several channels. Addressing is done by channel, not by target processor. Messages are received in the order sent.

To the user, communication takes the form of a virtual circuit, a logical full-duplex connection between processors. A virtual circuit can be modeled as a box with four ends: DIN and DOUT control data transmission, and CIN and COUT control transmission of control information. Circuits can be spliced together, or attached to devices. Circuits can also be joined into groups.

A virtual circuit is set up in the following way: a socket is created with a globally unique name, and processes then request connections to the named socket. Routing information is implicit.

Virtual circuits support location transparency, since sockets can move, but not replication transparency. If a circuit breaks, it is set up again, although it is up to the user to handle recovery of state information. Machine failure will destroy a virtual circuit.

A transparent distributed file system was set up. When a remote file is accessed, a socket name is generated, which is used to establish a connection with a daemon at the file server processor. The daemon carries out the file access on behalf of the remote user.

In conclusion, offloading of tasks was found to work well. The path manager maintains very little state. The splice interface between virtual circuits was found to be very efficient, although UNIX scheduling was not appropriate for fast setup of circuits.

2.5 Andrew Birrel: Grapevine

Grapevine is a mail transport system, providing naming, authentication, access control, and resource location services. The naming service associates to each individual a mailbox site, a network address, and a password. To each group name is associated a list of names. The authentication service accepts a name and a password, and returns a boolean. The access control service accepts a group name, and an individual name, and returns a boolean. The resource location service accepts a service name, and returns the name of the "nearest up" instance of that service.

Names are stored in registries. Registries may be either geographic, or organizational. Registries require partitioned and replicated data bases of names, and a centralized allocation of registry names.

4.2 Michael Schroeder

The topic of this presentation was authentication and access control. It was argued that it is necessary for authentication and access control to be implemented at the lowest level in a system and be universally applicable. Otherwise, each application would have to design its own mechanisms and the problem would perpetuate itself. Authentication/access control is related to the issues of naming and resource location. The approach taken is to use the same database for the data objects and authentication/access control.

The authentication is done by an "authenticator". A client presents his name and the name of the intended communicant, and gets back an authenticating bit pattern. The protocol implementing the authenticator function is described in detail in [CACM, December 1978, pp. 993-999].

Schroeder has written the encryption algorithm in software. It was very slow. Therefore, a password scheme was used instead in the actual Grapevine system.

Schroeder concluded that it is unsatisfactory to provide only encryption or only authentication. To build a system right, everything must be encrypted in a continuous manner. A hardware encryption chip would be required. Encryption cannot be an option but must be used at all times by everybody.

Discussion:

Q: Why do you not use a trivial encryption algorithm now and use an encryption chip later when it is available?
A: Even a trivial algorithm can affect performance.

4.3 Dave Gifford

This presentation concerned key generation and encryption techniques. Keys should be random, i.e. unbiased and uncorrelated. To achieve this, a biased uncorrelated Bit Generation by a noise diode was used in a key generation board built at Xerox PARC. Then the bias was removed by software.

The encryption board used for the Dolphin processor is fast; therefore encryption is (nearly) free. The board provides the four operations: Encrypt, Decrypt, Checksum, Random Bit.

4.4 Bruce Lindsay

Bruce Lindsay has been working on distributed databases built on communicating mainframes. Important issues in distributed databases are site autonomy and graceful growth. Site autonomy implies independent control of database sites, stand-alone operations, and local control over local resources and data. Graceful growth implies the avoiding of name conflicts (a unique site id can be part of the name, for example), and growth by accretion. The latter implies no nationwide sysgen, and intersite sharing by bilateral agreement only.

Distributed Authorization:

The following outlines a distributed data base protection proposal being considered for R*, a distributed version of System R, now under development at IBM San Jose. Authorization is done by access lists. Access may be "grantable". Revocation is recursive: i.e. if an access is revoked from a user A, it is also revoked from all users who have received the access right from the user A. Authentication and authorization to local data is locally enforced. Thus, authorization decisions can be made locally and accesses can be revoked locally.

Authorization is granted on the basis of the unique id of the accessor, i.e. USER SITE (where the site-id is unique). Note that objects using other objects need authorization. Authentication is only done at a site-to-site level. There is no point in using a finer grain. Authentication should not be made on every request. A grant of authorization has the following form:

< user1 site i : object ---> user2 site j : object >

Example: The owner, Bruce N1, of a table T has an authorization entry for himself, i.e.

<Bruce N1 : T ---> Bruce N1 : T>

at the site of T. Bruce N1 can then grant access to the table T to another user, e.g.

<Bruce N1 : T ---> Jim N2 : T>

Object Authorization to Objects:

In the concept of database "views", a "virtual" table is defined by a query on a "base" table, e.g.

```
DEFINE VIEW notrich N1 AS
SELECT name FROM employee
WHERE salary < 10000;
```

The view definer must be authorized to access the base relations. The view may be granted to users who do not have access to the base relations.

Example: Continuing the previous example, we have at site N1

```
<Bruce N1 : T ---> Bruce N1 : T> {initial grant}
<Bruce N1 : T ---> Jim N2 : T>   {grant to Jim}
```

Now Jim defines a view by

```
DEFINE VIEW V N3 AS SELECT * FROM T;
```

If Jim at site N2 wants to grant the view to another user, say Robin N4, then the following entries must be stored at site N3:

```
<Jim N2: V ---> Jim N2: V>      {initial view grant}
<Jim N2: V ---> Robin N4>       {Jim's grant to Robin}
```

4.5 Jerry Popek

This session was included in the workshop because the issues raised here are important ones. It has often been pointed out that it is difficult to add security and protection mechanisms to a system once it is built. We believe that some significant level of security will be desirable for many applications in the near future. Therefore knowledge of suitable approaches needs to evolve to the point where the system impact of the goal of secure operation can be taken into account early in design and development.

5. Applications

Chaired by Richard Rashid

This session could have been labeled "Goal Oriented Systems". Emphasis was placed on how specific application systems were influenced by the problems they were trying to solve. Each speaker was asked to address not only the nature of the system he or she had built, but also to describe characteristics of the problem which had led to particular design and implementation decisions. In addition, the speakers were asked to comment on the key attributes of their systems which lent coherence to their work.

Four presentations were made during the one and one-half hour session:

Roy Levin talked about the Xerox PARC grapevine system - a mail system designed to serve a large user community.

Carol Powers of TRW discussed real-time embedded systems primarily designed for aerospace applications.

Clarence Ellis dealt with office systems being designed at Xerox.

Robin Williams presented a distributed database system being built at IBM.

As can be seen from this list, the range of applications discussed during the session was as large as time would permit. The goals of the participants in building their distributed systems varied widely and the systems they built reflected these differences.

5.1 Roy Levin: Grapevine

Grapevine has as its goal mail delivery for a large, physically dispersed user community. Specifically, the system must provide adequate mail security, a high degree of availability, and adequate performance for use by human beings. The mail delivery function of Grapevine does not require, however, critical real time performance or even complete consistency of the distributed mail database. As long as the system will act on a request for message transmission in a "reasonable" amount of time (minutes to hours) the user will be satisfied. Likewise, atomic updates to a distributed database are not required. It is sufficient that the components of the data base be guaranteed to converge eventually to a consistent state.

In some cases, even non-serializable semantics are acceptable. For example, data base components keep a record of added and deleted names. This list is used to detect duplicate requests. To avoid keeping arbitrarily long lists, the list is periodically shortened, allowing for the small possibility of some duplications. The frequency with which the list is shortened is a function of the relative cost to the system of long lists and the cost to the system's human users of duplicated requests.

Grapevine provides two basic services: naming and message management. Name service includes registration and authentication. Message management consists of buffering and delivery. The mail service accepts messages from users, transports copies of messages to temporary storage, and retrieves the copies on request. The issues that arise in the design of the mail service include determining appropriate locations for buffering messages, the elimination of duplicates, and message sequencing.

The repeated use of message transmission and registration in various components of the system lends coherence to the system and reduces its logical complexity. Grapevine makes use of its own message delivery service in the maintenance of its registration database. Likewise registration is used to keep track of Grapevine's component parts such as printers and file storage devices.

5.2 Carol Powers: Real Time Embedded Systems

The problem domain addressed by Carol Powers contrasted sharply with that of Grapevine. Real-time response is absolutely critical, but the systems are simplified by the fact that they are typically constructed as "closed systems". It is usually feasible to re-initialize or "sysgen" an entire system when the problem domain or situation changes and thus such tasks as name location, error checking, access control and self-defense can be performed statically.

Whereas Grapevine's solution depends on loose bindings between services and servers, the systems Powers described require such high performance that real-time name lookups are usually unsatisfactory. Decisions about location and replication of data can also be done statically.

The principle engineering problem faced in the design of these systems is deciding which tasks are best allocated to the operating system and which are application dependent. The "end to end" argument is frequently applicable to these systems since a high level of error detection and recovery is often natural to the application. This "end to end" argument dictates that few if any checks should be performed by the operating system, for example, in the delivery of messages between system components. Missing messages are often best handled in the application.

These systems are typically built using a large number of small components which communicate via messages. This allows a modularization of implementation and design essential to a production environment.

5.3 Clarence Ellis: Office Information Systems

Clarence Ellis described both research performed by Xerox in the area of office organization and office tasks and a system under construction to automate office functions.

Current office organizations are distinguished by the fact that people are very good at dealing with informal problem solving, bargaining and exception handling. A study of a billing office, for example, revealed that workers there tend to view themselves as "creative problem solvers". Much of the work they perform involves fixing errors: applying previous payments to compensate for errors, changing payment plans, and co-ordinating these actions with the credit department.

One approach being studied at Xerox to solving the office automation problem is the use of active "forms" which embody both declarative and procedural information much in the fashion of Smalltalk objects. Ellis emphasized, however, that there are no simple answers to the question of how to automate the modern office. The necessary tools must come from a number of disciplines including Artificial Intelligence, Operations Research, Computer Science, Psychology and Anthropology.

5.4 Robin Williams: A Distributed Database Transaction System

Robin Williams described a distributed database transaction system. The system consists of a collection of (possibly specialized) machines communicating through a network. All communication is done by explicit message passing. To maximize availability and reliability, the distributed system appears to the user at a terminal to be a single entity. An advantage of this type of system is its potential for easy growth. When it becomes overloaded, it is possible incrementally to add new machines to the collection.

The principal performance problem is communication. Off-loading a task onto another processor is not cost effective if the additional communication overhead is greater than the original task. When a machine sends a message to another, should it switch tasks? If the reply is immediate, for example, it is more cost effective not to switch. This suggests that it might be useful to introduce two kinds of messages: short messages that do not cause task- switching and longer messages that do.

Shared memory can significantly increase communication throughput, but this advantage must be weighed against a potential loss in reliability and/or availability.

System D was discussed as an example of a distributed transaction system. It consists of a collection of IBM Series 1's connected by a ring network. Each processor has a resource manager which contains a database of local resources, detects failures and reallocates resources.

6. Naming

Chaired by Jerry Popek

The speakers were asked to use the following questions as a set of guidelines in their talks:

Are low level unique names necessary/possible? Under what conditions?
How does one generate low level unique names?
How does one generate high level unique names?
Is name rebinding necessary/feasible on a large scale? How is it done?
Implementation issues: caching, context dependent interpretations, etc.

6.1 Butler Lampson

To generate uids, there are two major approaches:

1. Central service. This approach is not generally satisfactory.

2. Decentralized service - classically done by prefixes. Telephone numbers are an example. Such a name should be able to grow in both directions infinitely. One approach is for each piece of the service to dole out names from a partition. Sometimes, one may need a new partition. In this case, a problem arises when semantics have been assigned to pieces of names, geographic meaning, for example. Every field should be essentially of infinite length. Applying semantics is difficult in fixed length fields.

DEC proposed using a white noise random number generator for processor id's on the Ethernet, so that we would never have to go to a central source. This is fine, but one should not make the whole system depend on the uniqueness of such numbers, because duplicates are difficult to discover. Unique processor numbers are supposed to make reconfigurations easy.

What are names for? A name is something that is mapped into something else or bound to another name. Frequently we go from a variable length name to a fixed length name.

Is a mapping constant? If not, this is called rebinding. In this case, caching is hard. If the mapping is constant, caching the result is easy.

Consider the following situation where the context is the set of global variables. We are opening a file named "Foo".

```
P = Begin
  .
  .
  .
  Open ("Foo")
  .
  .
  .
  End
```

To control the binding of "Foo" explicitly, we may want to specify the context as a formal parameter of P:

```
P(workingDir: Directory) = Begin
  .
  .
  .
  Open (workingDir, "Foo")
  .
  .
  .
  End
```

We could even pass the procedure "Open" as a parameter.

```
P (Open: OpenProc) = Begin
  .
  .
  .
  Open ("Foo")
  .
  .
  .
  End
```

The question is: Should the context be explicit or implicit; i.e., should basic properties be parameters or part of the environment? The advantage of leaving the context implicit is that there are some things you don't want people to think of as variable. The parameters should be those items that people think of as variable. On the other hand, if you really want to know what is going on, things should be explicit. One can make a context explicit by turning globals (free variables) into parameters. Another way of achieving this goal is by an explicit mapping of context-dependent names into unique names; the unique name is used from then on.

The way you get rid of the parameters is by binding. There is a question about when binding is done. If binding is done and the object containing the name is then moved to a different context, the name will get you to the same object. If binding is not done before a move, the name may be bound to different objects in different contexts.

6.2 Hugh Lauer

Hugh Lauer addressed two problems in the area of naming. The first was that of including a naive participant in the process of fundamental binding/naming operations of the system. Permitting such participation in practice is more difficult than it may seem, as is illustrated by the following experience. Early in the development of the Pilot operating system, it was decided to name processors uniquely, and also to provide unique low level names for objects. Unique, universal name generation depends on unique processor names (assigned at the factory). But, the factory wasn't set up to provide them. First, it was discovered the people in the factory (the naive participants) were updating the processor number on new versions of the processor. An attempt was made to replace all those boards that had incorrect numbers, but the manufacturing people still couldn't respond. Finally, it was discovered that the Dolphin processors were all being created with the same processor id!

The second problem is the following: given a namespace of fixed size with names chosen freely but uniquely from within the namespace, merge two such namespaces such that the names are still unique and the same size. (Interconnecting two previously disjoint networks is the obvious case.) There are two solutions. First, avoid the problem by allocating from a naming authority spanning all the universe. This solution is costly and may require the involvement of a naive participant. Second, invalidate instances of duplicated names, forcing rebinding to new names. The cost of rebinding may be so high that it must not be allowed to happen. Thus neither approach is entirely satisfactory.

6.3 Bruce Lindsay

Bruce Lindsay addressed seven requirements for the user sensible names (called "print names") in R*, the distributed version of system R.

The same name can be used by different users to refer to different objects.
Different names can be used by different users to get to the same object.
A name instance may be resolved differently at different times. (Testing, for example.)
Global name resolution information is not required at any one site.
Names may be context independent.

Names may be context dependent.
An object's representation may migrate without changing names.

6.4 Discussion

Reed: So far, all the problems presented are problems in centralized systems.

Lampson: with new ways to mess up.

Reed: There seems to be a fundamental issue - we assume we can simplify things, and then rediscover the same old problems.

Lauer: In centralized and small distributed systems, we think about naming a little and then stop. The difference is in the scale of the problem rather than in the quality of the problem.

Cheriton: A name is a replacement for a description. A certain amount of ambiguity is available in such a description. You are always going to have ambiguity, and perhaps we should be dealing with it.

Shoch: We must consider the distinction between human sensible names and machine sensible names.

Lauer: Remember that one man's name is another man's address. Also, machines are good at comparing names, while humans are good at resolving names.

Schroeder: People want context sensitive names and descriptions.

7. Communications

Chaired by Butler Lampson

In the sessions on Communication, a number of communication schemes were described, some measurements were reported, and some philosophical comments were made. Everyone with a scheme followed the outline below in describing it (though the points of emphasis varied greatly):

Semantics
  Synchronization (Wait on send? Multi-port receive?)
  Exceptions

Failures
  Loss and duplication
  Crashes

Parameters and types
  Authentication (Type-safe? For abstract types?)
  Encoding (Of values? Of references? Extensible?)

Connectivity
  Who talks to whom?
  Replication

Basis
  Built on what?

Performance
  Latency
  Bandwidth

Resource allocation (no one had much to say about this)
  Fairness
  Congestion control
  Accounting

7.1 Richard Rashid

Richard Rashid's presentation concerned an existing system, which has been running since March, the SPICE/DSN distributed operating system. It is a message-based system which contains two kinds of entities: messages (a collection of types and data), and ports (a port can be regarded as a queue of messages or an address where messages are sent). The port names are not known directly by processes, but through a local name table (similar to a capability list).

Synchronization: Ports are finite length queues. The sender is normally not suspended on send. If the queue is full, suspending the sender is only one option. A sender may suspend until the queue has room, or return with an error indication, or receive a later notification (in the form of a message) when the port can take another message.

Exceptions: Exceptions are signalled across process boundaries with high priority "emergency" messages. Software interrupts may be invoked by message reception.

Loss and duplication of messages: Messages have a type field that indicates service requirements, e.g. reliable transmission, sequentiality, maximum age, priority.

Crashes: The system has complete knowledge of connectivity and sends an emergency message to each processor which has "send" access to a destroyed port. (No distinction is made between a partitioning and a node down: in both cases, all port connections involved in the failure are broken.)

Authentication: Messages contain collections of typed data. The typing system is "safe" only for port values.

Encoding: Port rights could be considered references. The data typing system is user extensible and used primarily for machine-machine communication where machine data representations vary.

Connectivity: All processes begin life with two ports: the kernel port and the data port. The father of a process may get access to kernel and data ports of children. Port access rights (send, receive, ownership) may be passed in messages. Processes may send to only those ports which are in their access lists (maintained and protected by the kernel). Send rights may be replicated without restriction.

The basis of the system is currently VAX/Unix. An implementation on Perq is in progress. Lisp, C, and Pascal are supported as languages. Performance (under the third release of Berkeley Unix): The total time for sending and receiving a 1024-byte message within one processor is approx. 5 msec, of which 2 msec are used for scheduling time in Unix, 2 msec to copy data in and out of the kernel, and 1 msec in the IPC code. As 65% of the latter is used for entry/exit, the actual message communication takes approx. 350 microseconds.

7.2 David Cheriton

David Cheriton also discussed a message based system. The base architecture under consideration is a set of computer modules (i.e. processor and memory) communicating over a fast, reliable bus, and a message kernel of a uniprocessor machine. The design philosophy can be characterized as Occam's Razor: all things being equal, pick the simplest solution. Therefore, a major concern is what ideas to exclude from the design. A second concern of the design is efficiency. Cheriton proposed the following "Conservation of Elegance Law": inefficiency at one level breeds kludges at the next level.

Message-Passing made Simple:

Consider a message-passing system the semantics of which are expressed in two different views. The sender's view of message-passing is that of a procedure call. The reasons for this view are that it is the simple, familiar semantics, it is commonly needed, and it is efficient to implement. The receiver's view of message-passing is that of incoming, queued messages needing replies. The reasons for the receiver's point of view are the same as stated above for the sender's view. The "asymmetric" semantics is justified mainly by experience. It fits well the server/client model, and also seems to handle pipes, or real-time systems. It seems like the natural way of looking at it.

Thus, communication between a sender and a receiver is performed in the following way: The sender sends a message to the receiver and then blocks. After some time, the receiver will accept the message, perform the corresponding actions, and send a reply message to the sender. At reception of the reply message, the sender is unblocked. Note the following characteristics:

Messages are queued until explicitly received. Receivers may block for messages.
Replies can occur in a different order than messages were received.
The receiver can control the sender for synchronizations, flow control, and exceptions.

Concurrency is achieved by having many processes. The typical model is a "manager" process who has several "helper" processes work for him. The manager process only receives and replies to messages; it does not send messages, and therefore it only blocks when there is nothing to do. In effect, the manager/helper scheme implements an asynchronous "send-message" mechanism. Cheriton believes that if a process can do some asynchronous activity after the call (i.e. the send), it is natural to split this activity into a new process (i.e. the helper process). A different process must be created to handle time-outs after the send.

In addition to the simple message-passing mechanism described above, a message forwarding scheme has also been implemented. In this scheme, the sender includes in the message not only the id of the direct recipient of the message, but also the id of a process to which the message should be forwarded. The reply message is sent only from this second receiver to the sender. Examples of uses of this message forwarding scheme are: name servers; exception servers; a "hunt group" virtual call interface; message reprocessing; printer servers.

Communications: Control and Data

There are two different kinds of messages: Many small, fast control messages, and fewer, larger data transfers. Thus, the system implements 8-word messages plus a separate "virtual" DMA transfer facility. The DMA transfer calls have the following form:

Transfer-to (destination-id, destination, source, count)
Transfer-from (source-id, source, destination, count)

Example: File sever request. The user sends a Request (control) message: "Send me N bytes from file F, offset K into my space starting at address A." Then, the file server transmits data directly into the requesting process' space (as in DMA).

Conclusion:

The semantics of this mechanism are: Synchronization by "send and wait for reply"; a single port per process; global port identifiers. If a port is non-existent, an error return occurs. There is no support to handle crashes. Authentication and encoding are not done. The system is fully connected, i.e. anyone can talk to anyone. As a basis, a low-delay, high-bandwidth network is assumed. Resource allocation is up to the receiver.

7.3 Richard Watson

Richard Watson described a message system that is running on a Cray 1. The network aspect of it is not yet implemented. Watson's group is trying to build a distributed operating system in a layered way, so that it can be used on top of other operating systems. The intent is to use this distributed OS as the resident OS in the Cray 1. The basic interprocess communication comes in two levels: The high-level request/reply structure conventions with typed parameters, and the low-level uninterpreted arbitrary-length messages.

Semantics: The operations defined on the arbitrary-length messages are "send", "receive", "abort", "wait", and "status". There is no notion of emergency or interrupt messages. There is also no notion of opening connections. The user can obtain time-outs as exceptions.

Failures: The system protects against missequencing. Crashes are detected, but recovery is a higher-level issue and is not considered in the message system.

Parameters and Types: Authentication is done by capabilities. Parameters in messages are typed.

Connectivity: Any process can communicate with any other. Alternate routes are provided.

Basis: The system should support a variety of technologies. We have made no assumptions about a specific machine.

Performance: Latency is on the order of microseconds or tens of microseconds. Bandwidth is in the range of 9,600bps to 50 mbps (on the physical channels).

The basic philosophy for resource allocation is overcapacity.

The major differences from other message systems are the possibility of arbitrary-length messages and the simpler notion of a port. As far as user processes are concerned, there is no difference between local and remote communication. The system provides optimization for the local case.

7.4 Barbara Liskov

Barbara Liskov discussed some communication primitives to be used as part of a programming language. The programming language is intended to support the organization and execution of distributed programs for "office systems". None of the described primitives has been implemented.

The entire system is comprised of guardians which are the unit of locality, i.e., each guardian is wholly contained on one node. Each guardian contains a number of processes. There are two different kinds of calls: Calls within a guardian are normal CLU calls, with sharing of arguments between the caller and the called procedure. Inter-guardian calls use the following remote procedure call primitive:

```
call P(a1,.....) on x
        when R1(f1:T1,....) : S1
        ---
        ---
        when failure (s: string) : Sf
        when timeout <time>: St
        end
```

Within a guardian, calls to that guardian are received by executing the receive command. This has the following form:

```
receive on x,y,z
        when P(f1:T1,...): ...[commit or abort] reply Ri(...)
        ---
        ---
        end
```

Explanation:

P(a1,...) is a message, where P specifies a command and a1,... are the arguments.

x specifies a port, i.e., a buffer where a message can reside until it can be received. (Ports are the only unique low-level names in the system.)

R1,... specify user-defined possible replies from the receiver.

All parameters are typed. Message communication is strongly typed also in the sense that the port types must list the types of messages that can be received at that port.

"Failure" and "time-out" are system-generated replies. "Failure" occurs, for example, if the port has been destroyed. The argument string specifies the reason for the failure. This information is useful for debugging.

Parameter passing is done by value. It is possible to send names in messages, but, except for port names, these names cannot be used for anything else but to send them back to the caller, who "understands" the names.

This remote call is more than just a pairing of messages. The desired semantics is "exactly once executed" in case of the return of a reply, and "no execution" in case of a failure or time-out. In the applications which this programming language should support, atomicity is really needed. Therefore, each remote call in our scheme is an atomic operation.

Discussion:

Q: The mainstream of research seems to try to cover up the differences between local and remote calls for the user. You introduce two different calls for these cases. Why?
A: We felt it was important to distinguish actions that could be done locally from actions that were remote and therefore likely to be expensive.

Q: What is the essential difference between this scheme and a scheme in which the remote unit simply provides procedures for remote call?
A: Ports and the receive statement permit the guardian to do scheduling.

7.5 Bruce Nelson

Remote procedure call is a programming language primitive intended for use in distributed systems. A program on a client machine may invoke a procedure executing on a server machine. Remote invocation appears exactly the same as a local procedure invocation. In the absence of crashes or communication failures, the requested action is performed exactly once. In the event of failure, the action may be performed several times, but the caller will receive the results of the last invocation. The remote procedure call primitive performs type checking and authentication. It is intended for use in a homogeneous environment: specifically, the Mesa language in a system of Dolphin and Dorado processors.

A number of performance measurements were taken for remote procedure calls varying a number of parameters. The parameters varied include differing numbers and types of arguments, using different protocols (including no protocol), doing or not doing process switching, and doing or not doing software checksums. By building a network interface that directly supports remote procedure call, dramatic increases in performance were observed (from 100 milliseconds to about 800 microseconds on the Dolphin).

7.6 Dave Clark

Performance measurements of the TCP and Internet protocols were undertaken on the MIT Multics system. A null file was sent between two asynchronous processes under a variety of conditions. The following conclusions were drawn from the measurements. The proportion of cost associated with the details of TCP (e.g. header formats, out of order packets) is small. The principal cost is associated with gross structural considerations, such as checksums and copying data. There is no single area in which all the cost is concentrated. For file transfer, it is best to optimize the inner loop where the packets are formatted. For Telnet, the overhead costs should be minimized. The costs on Multics were dominated by scheduling costs such as inter-process communication, timers, and process swapping. Since Multics is a nine-bit machine, additional costs were incurred in conversion to the eight-bit format used by the protocols.

7.7 Jerry Saltzer

The end-to-end argument states that certain functions can be completely and correctly implemented only with the knowledge and help of the application standing at the ends of a communication system. Such functions cannot be provided by the communication system itself, although the communication system might provide partial support for the function to enhance performance.

Reliable file transfer in a distributed system is an example. There are many opportunities for error in the course of transferring a file from a disk belonging to one machine to a disk belonging to another. The medium on which the original file was stored may have become defective, a processor may encounter a transient error, the communication system may drop or change bits or lose or duplicate packets, or either processor may crash in the middle of the transaction. Finally, there may be software errors at any point in the transfer.

One way to engineer this system is to attempt to make each piece very robust. Unfortunately, piece-wise protection tends to leave gaps, particularly if one has to deal with software errors. A better way to insure correctness is to have the file transfer software use a (large enough) checksum to compare the transferred file with the original. If the comparison indicates that something is wrong, the transfer is re-attempted. By making (for example) the communication protocols very robust, it is possible to lessen the number of retries, but an application-dependent end-to-end check is still required to insure ultimate correctness.

8. What Do We Need from Theory?

Chaired by Susan Owicki

Leslie Lamport gave us the definitive statement of the role of theory in distributed systems (and elsewhere), courtesy of the Oxford English Dictionary: a theory is "a body of theors," and a theor is "an ambassador or envoy sent on behalf of a state, especially to consult an oracle or perform a religious rite." Other opinions existed, however, and the question of the relationship between theory and practice provoked considerable debate.

The first speaker was Roger Needham, who described himself as a non-theoretician. He believes that the chief contribution that theory can make to those who build systems is to provide models that allow the builders to reason in an orderly way about the correctness of their systems. He cited three examples from his own experience where such systematic reasoning seemed desirable but impractical without more theoretical foundations. One case was a protocol developed with Mike Schroeder. The problem was a complex one, due to timing interactions and failure modes, and numerous versions of the algorithms were tried before one was finally selected. Each time a new version was proposed it was examined by a number of people who feverishly looked for errors. The process gradually converged on an implementation that seemed to be right. But there was no systematic way for the designers to convince themselves that all errors had been detected. Other examples concerned circular reasoning about circular programs (the Grapevine mail server uses its own message system to maintain its databases) and reliable systems in which a basically simple method must be expanded to cope with failures. The latter systems suggested a lurking theorem to the effect that the new system provided the same services as the simpler one, but it was not clear how to state or prove such a theorem.

Mike Fischer identified two goals for theory: it should establish a framework for cataloging knowledge and provide insight into general phenomena. The methods for accomplishing these goals are abstraction, which allows the theoretician to throw away excess baggage in the form of irrelevant detail, rigor, and generality. If the theoretician makes the right abstractions, he can draw conclusions that are rigorous, widely applicable, and still relevant to the practical problems which were his starting point. Of course, the trap in theory is abstracting the wrong parts of the problem; the results then are uninteresting because they are no longer relevant. The theoretician is like a scout looking over new territory; he travels light, and can explore a wider area than a system builder who must always keep in mind the work of implementation. (This last point provoked considerable disagreement. The opposing view was that builders are the ones who discover new territory, and then theoreticians can come in and examine all of its aspects. Butler Lampson's summary was "practitioners explore, theorists pave over.")

Ed Lazowska discussed analytic modelling as a tool for performance analysis, considering the problem characterized by David Cheriton as "How do you make a dinosaur run faster?" He stressed that performance must be designed into systems; it cannot be an add-on feature. So we need tools for analyzing performance before the prototype stage, and analytic modelling provides them. Such tools allow one to evaluate more options than could ever be built and measured. Modelling allows one to concentrate on better solutions at a high level, and avoid optimizing bad approaches. However, analytic modelling cannot be subcontracted out -- it is essential that the designer be a participant in the process if the results are to be at all meaningful. The big problem, as characterized by several members of the audience, is that the load placed on complex systems is so hard to predict that only very approximate modelling may be worthwhile.

The last speaker was Leslie Lamport, who pointed out that theory is "that part of a subject concerned with its fundamental principles and methods" (also courtesy of the OED.) Much of theory concentrates on principles, with an emphasis on telling you what you can't do. But methods are also extremely important. Proof methods provide means of reasoning about programs, and algorithms provide basic patterns that can be used in building more complex systems. An important piece of advice, much ignored by both builders and theoreticians, is "state the problem before giving the solution." David Cheriton commented that builders tend to know of a problem in a very uncertain way; they identify it by groping around and tripping across it. Theoreticians build a framework of problems, distilling and saying something more rigorous.

In the general discussion that followed the presentations, we groped around the problem of what theoreticians and builders can do for each other in computer science. Physics was proposed as a model, but most felt that physics is fundamentally different in that its phenomena are given in the world, while computer science creates its own. Ultimately, what theory has to offer is the clarification that comes from ignoring

irrelevant detail; its weakness is that sometimes that detail turns out to be essential. The only solution is to evaluate the results, and try again if they don't match reality.

9. What are Important Practical Problems?

Chaired by Barbara Liskov

The official title of this session was "Important Practical Problems" (to balance the preceding session on theory), but the actual discussion was broadened to include some topics that had not yet come up. Three topics were discussed:

What are the important practical problems in distributed computing?
What are the fundamental issues in distributed computing?
What is different about distributed computing?

This session included a talk by John Shoch on the topic, "What is Different About Distributed Computing?" Shoch wrote up his talk as a short paper, which is included as Section 10.

The program committee had some preconceived notions on the second of these questions, and we organized around the areas we felt were most important. We also organized some sessions, namely the ones on real systems and applications, with the goals both of exposing additional issues, and of testing the defined issues against real experience.

One important issue that was not addressed was synchronization. Personally, at the time the workshop was organized, I thought synchronization was not much of a problem. In my work, we are providing read/write locking tied to a transaction system, and I expected that to be sufficient. I have since discovered that the problem is not so simple. Highly concurrent algorithms may be needed, and these require more flexible locking rules than our automatic locks provide.

As far as the difference between distributed and centralized computing is concerned, I have not yet discovered anything fundamental. Every problem I have studied in the context of a distributed system has an analog in a centralized context. Often, however, the problem has not been solved satisfactorily in the centralized context. An important example is construction of fault-tolerant software. The major difference between distributed and centralized systems appears to be the algorithms needed to achieve such things as fault-tolerance; the distributed algorithms tend to be significantly more complex than their centralized counterparts.

A problem that can arise in writing in distributed systems is that the nature of the solutions can lead to confusion between the logical and physical aspects of a task. This confusion was apparent in the session on naming. For example, a unique name space must be logically centralized or the names won't be unique. This does not prevent it, however, from having an implementation that is physically distributed.

Comment: One fundamental characteristic of a decentralized system is the absence of any global knowledge.

Saltzer: Concerning the discussion of fundamental differences between centralized and distributed systems, I have a feeling of *deja vu*. One can compare this situation to the difference between time-sharing and batch systems. The insight that was useful then was that the solutions to time-sharing problems also applied to batch systems, but the problems had been latent in batch systems and had been solved in an *ad hoc* manner.

Lindsay: One important issue in decentralized systems is the notion of partial failures. This idea will also be useful in centralized systems.

Watson: I see the following characteristics of decentralized systems: more heterogeneity; distribution of state; distribution of control; communication via messages. What we need are new tools, new mechanisms, a deeper understanding of the issues involved, and a notion of how much of the distributedness should be visible.

9.1 John Shoch

The following are some practical issues associated with distributed computing:

1. Installation and configuration management: This is a pragmatic but very real issue, which may dominate some tradeoffs among technical alternatives. For example, the choice of a 48-bit flat address space in the Ethernet specification is meant (in part) to simplify the process of installing and moving machines.

2. Performance and performance debugging: It's not enough just to show that a system works -- you want it to work *well*. (I first heard the term "performance debugging" from Bob Sproull; does the term have deeper roots?) For example, we have seen some instances where communications protocols were used for a file transfer which started out fine, suddenly slowed down, but eventually sped up and completed. The various algorithms were "correct," and the transfer ran to completion using the recovery mechanisms, but the performance had crumbled.

In the networking area there's a whole collection of such situations, with colorful names: the "rack of bowling balls effect," "lightning strikes" that induce "doubletalk," etc. In developing the "worm" programs we also have seen systems which ran well, then went unstable for lengthy periods. Finding, replicating, and eliminating these kinds of bugs is very challenging. An area that cries out for work is the application of various types of adaptive control theory to the running of distributed systems.

3. Educating people about the change in paradigm: The field is changing very quickly, as the new paradigm of "distributed" systems gains momentum. Unfortunately, the level of education is not always as high as one might expect, and many of the issues which we address in the research arena remain as complete mysteries to much of the profession. Even worse, many of the words and concepts get bandied about, but often without a great deal of understanding.

9.2 David Reed

One important practical problem of distributed systems is the issue of autonomy. Systems are built by accretion. The issues here are physical connectivity (must be preplanned), the big software investment which makes it impractical to change software, and the problem of comprehensibility. In centralized system, it has been possible so far for single persons to understand the entire system (even of the size of MULTICS). This will not be possible for distributed systems. How can we comprehend parts of the system without comprehending all of it?

10. What's Different About 'Distributed Computing'?

or

The Change in Expectations About Computer Systems

John Shoch

One of the most popular phrases in contemporary computer science is the notion of "distributed" computing -- every imaginable kind of computer, peripheral, communications line, operating system, or program has been anointed with the magic descriptor "distributed." Any serious discussion of this phenomenon, however, frequently breaks down over the effort to define "distributed," as distinct from "non-distributed" or "centralized." Ultimately one confronts the question, "Is there anything really different about 'distributed' computing?"

The easiest answer is, "No, there is nothing different about 'distributed' computing." Here are several arguments in support of that position:

1. The word really has no meaning -- almost any system can be described as "distributed" and the term doesn't identify any meaningful, distinguishing characteristic. Six years ago David Parnas commented upon the way in which another buzzword became devoid of meaning: ".... the statement 'our Operating System has a *hierarchical structure*' carries no information at all." [Parnas, 1974]

2. The word has been co-opted by the marketeers, and used to label even the most mundane computer system. There's a great deal of truth to this argument: what used to be called a "key to disk data entry system" is now sold as a "distributed data entry system," or even as an "intelligent, distributed data entry system." In many ways, our language has failed us -- we have not been able to preserve the crisp meanings for words that make them so useful.

3. Maybe this is just a passing fad, a media hype, or a form of mass delusion and hysteria sweeping the industry. Perhaps it's like "est": you pay several hundred dollars to go to a seminar, and come back as a convert and true believer.

4. The intellectual problems are no different than the kinds of things being tackled in traditional computer systems. If you burrow into any large operating system or application one can find problems similar to those now attracting interest.

Each of these arguments has some merit, and I'm sure that others could expand upon this list. Nonetheless, I do believe that the answer to our question should be that there *is* something different about distributed systems. (Some would argue that I have a vested interest in this position, however, since I have taught a course entitled *An Introduction to Distributed Computing*.) Here are some (tentative) manifestations that might lead us to this conclusion:

1. We can seek out some empirical evidence by looking at the group of people who attended the recent Workshop on Fundamental Issues in Distributed Computing. Several dozen experienced, thoughtful researchers responded to a call for papers on the subject of distributed computing, and came prepared to discuss pieces of their work which fell in this category. It is just not reasonable to believe that they have all been fooled by the Emperor's nudity.

2. Distributed systems seem to be characterized by a lack of central control (or what Jim Gray has described as global knowledge). There is no simple distinction between a "centralized" and a "distributed" system, but rather we see a spectrum of design points. A centralized system may still have race conditions, timing uncertainty, and inconsistency, but the problems can usually be managed adequately; distributed systems, though, cluster towards the other end of the spectrum -- some of the same problems emerge, but they may become more visible. Physical distribution of components, however, is only the first step towards a distributed system; the next step is to build the logical procedures required to run them in some coordinated fashion.

3. A quantitative difference in the number of processors does lead to a qualitative difference, and harnessing 100 autonomous machines produces a system which is different from a single machine which runs 100 times faster.

4. Finally, our expectations about how a "distributed" system should perform are very different. (Forest Baskett has noted that a train and a parking lot full of automobiles can both transport a group of commuters -- but we worry a lot about the scheduling and efficient utilization of the locomotive, while the cars remain idle all day.)

It is this last point which is, I believe, most important. (This might be characterized as the "inverse Jerry Brown argument" -- our expectations are not being lowered, but continue to rise!) Consider some examples:

1. Failure modes, rates, and recovery. It used to be that if a bit failed in memory the whole machine was deemed broken. Over time, we added parity and memory correction, so that the machine could continue to

function with a bad bit in memory. Yet even today, a bad interface or an error on an internal bus may cause a "machine check," and the whole system stops.

One interesting example emerged while measuring the performance of the Ethernet local network: we found several interface boards with severe hardware glitches, causing perhaps 1 out of every 100 packets to be garbled; yet when we mentioned it to the machine's owner he said, "Leave it alone, it works fine!" This kind of failure rate in an ALU would be catastrophic; it worked fine here because a full set of higher-level protocols had been built to overcome exactly these kinds of failures in this distributed system.

In a related example, Butler Lampson has on occasion tried to compare the code required to run a disk vs. the code used to run the Ethernet; I usually respond that a simple disk driver doesn't have to time-out the drive, or contend with duplicate replies to a read request! But as disk controllers become more sophisticated (and more independent) exactly this kind of checking now has to be introduced.

2. Overall system availability. With a large, centralized facility users expect that there may be crashes, or that the whole machine may be taken down for maintenance. In a distributed system, in contrast, we expect that most of the pieces will continue to run, even if one piece is disabled.

3. Impact of other users. People who use a shared, centralized system also come to expect that the behavior of other users will impact the overall system performance -- that's why so many of us learned to work at night or on weekends. But with more powerful independent machines tied into a distributed system we've come to expect that a whole machine will be at our disposal -- and it won't run any faster on Sunday morning!

Against this set of changing expectations we've been trying to define more capable systems -- and "distributed" systems are a fruitful approach. Thus, the same basic intellectual problems have been re-cast and re-defined, but in this new context; and these new problems help to make the field of distributed computing so challenging.

Unfortunately, I don't yet have a precise definition of a "distributed system," or what makes it different from any other kind of system.

But at times like this we can always appeal to a higher authority -- in this case the U.S. Supreme Court. The Court was confronted with a similar problem when trying to define obscenity or pornography, and it was Potter Stewart who invoked the famous observation which we can now apply to the definition of distributed computing:

I don't know how to define it, but I know it when I see it.

Acknowledgement

The thoughts in this brief note were developed during the course of the ACM SIGPLAN/SIGOPS *Workshop on Fundamental Issues in Distributed Computing*, December 1980; several conversations with Butler Lampson and Jim Gray were particularly helpful.

Reference

[Parnas, 1974] David Parnas, "On a 'buzzword': hierarchical structure," *Information Processing '74 (Proc. of IFIP '74)*, North-Holland, 1974, pp 336-339.

Appendix I - Attendees

Guy Almes
University of Washington
Dept. of Computer Science
Seattle, Wa. 98195

Gregory Andrews
University of Arizona
Dept. of Computer Science
Tucson, Ar. 85721

Andrew Birrell
Xerox PARC
3333 Coyote Hill Road
Palo Alto, Ca. 94304

Gregor Bochmann
Universite de Montreal
Departement d'IRO
Montreal, Quebec, Canada

David Cheriton
University of British Columbia
Dept. of Computer Science
Vancouver, B. C., Canada V6T 1W5

David Clark
MIT Laboratory for Computer Science
545 Main Street
Cambridge, Ma. 02139

Robert Cook
University of Wisconsin
Computer Sciences Dept.
Madison, Wi. 53706

Douglas Currie
Functional Automation/Gould, Inc.
3 Graham Drive
Nashua, N. H. 03060

Charles Davies, Jr.
6122 Escondido Ct.
San Jose, Ca. 95119

Clarence Ellis
Xerox PARC
3333 Coyote Hill Road
Palo Alto, Ca. 94304

Robert Filman
Indiana University
Computer Science Dept.
Bloomington, In. 47401

Piero Fiorani
Olivetti A.T.C.
10430 DeAnza Blvd.
Cupertino, Ca. 95014

Michael Fischer
University of Washington
Dept. of Computer Science
Seattle, Wa. 98195

Harry Forsdick
Bolt Beranek and Newman, Inc.
50 Moulton Street
Cambridge, Ma. 02138

David Gifford
Xerox PARC
3333 Coyote Hill Road
Palo Alto, Ca. 94303

Jim Gray
Tandem Computers, Inc.
19333 Vallco Parkway
Cupertino, Ca. 95014

Jerry Held
Tandem Computers Inc.
19333 Vallco Parkway
Cupertino, Ca. 95014

Andrew Herbert
Computer Laboratory
University of Cambridge
Corn Exchange Street
Cambridge, CB2 3QG
England

Maurice Herlihy
MIT Laboratory for Computer Science
545 Main Street
Cambridge, Ma. 02139

E. Douglas Jensen
Carnegie Mellon University
Dept. of Computer Science
Pittsburgh, Pa. 15213

Richard Kieburtz
State University of N. Y. at Stony Brook
Dept. of Computer Science
Long Island, N. Y. 11794

Leslie Lamport
SRI, International
333 Ravenswood Avenue
Menlo Park, Ca. 94025

Butler Lampson
Xerox PARC
3333 Coyote Hill Road
Palo Alto, Ca. 94304

Keith Lantz
Stanford University
Computer Systems Laboratory
Stanford, Ca. 94305

Hugh Lauer
Xerox PARC
3333 Coyote Hill Road
Palo Alto, Ca. 94304

Edward Lazowska
University of Washington
Dept. of Computer Science
Seattle, Wa. 98195

Richard LeBlanc
Georgia Institute of Technology
School of Information and Computer Science
Atlanta, Ga. 30332

Gerald Leitner
University of California, Los Angeles
Dept. of Computer Science
Los Angeles, Ca. 90024

Gerard Le Lann
INRIA, Projet SIRIUS
B.P. 105
78150 Le Chesnay, France

Roy Levin
Xerox PARC
3333 Coyote Hill Road
Palo Alto, Ca. 94304

Bruce Lindsay
IBM Research Laboratory
5600 Cottle Road
San Jose, Ca. 95193

Barbara Liskov
MIT Laboratory for Computer Science
545 Main Street
Cambridge, Ma. 02139

David Lomet
IBM Research Center
P. O. Box 218
Yorktown Heights, N. Y. 10598

Nancy Lynch
Georgia Institute of Technology
School of Information and Computer Science
Atlanta, Ga. 30332

Roger Needham
Computer Laboratory
University of Cambridge
Corn Exchange Street
Cambridge CB2 3QG
England

Bruce Nelson
Xerox PARC
3333 Coyote Hill Road
Palo Alto, Ca. 94304

Susan Owicki
Stanford University
Computer Systems Laboratory
Stanford, Ca. 94305

Jerry Popek
University of California, Los Angeles
Dept. of Computer Science
Los Angeles, Ca. 90024

Michael Powell
University of California
Dept. of Electrical Engineering
and Computer Sciences
Berkeley, Ca. 94720

Carol Powers
TRW
One Space Park
Redondo Beach, Ca. 90278

Richard Rashid
Carnegie Mellon University
Dept. of Computer Science
Pittsburgh, Pa. 15213

David Reed
MIT Laboratory for Computer Science
545 Main Street
Cambridge, Ma. 02139

Michael Rodeh
IBM Research Laboratory
5600 Cottle Road
San Jose, Ca. 95193

David L. Russell
Bell Laboratories
Holmdel, N. J. 07733

Jerry Saltzer
MIT Laboratory for Computer Science
545 Main Street
Cambridge, Ma. 02139

Fred Schneider
Cornell University
Dept. of Computer Science
Ithaca, N. Y. 14853

Mike Schroeder
Xerox PARC
3333 Coyote Hill Road
Palo Alto, Ca. 94304

John Shoch
Xerox PARC
3333 Coyote Hill Road
Palo Alto, Ca. 94304

Abraham Silberschatz
University of Texas
Dept. of Computer Sciences
Austin, Tx. 78712

Karen Sollins
MIT Laboratory for Computer Science
545 Main Street
Cambridge, Ma. 02139

Marvin Solomon
University of Wisconsin
Computer Sciences Dept.
Madison, Wi. 53706

Bjarne Stroustrup
Bell Laboratories
Murray Hill, N. J. 07974

Bruce Walker
University of California, Los Angeles
Dept. of Computer Science
Los Angeles, Ca. 90024

Richard W. Watson
Lawrence Livermore Laboratory
P. O. Box 808
Livermore, Ca. 94550

Maurice Wilkes
Digital Equipment Corp.
146 Main Street
Maynard, Mass. 01754

Robin Williams
IBM Research Labortory
5600 Cottle Road
San Jose, Ca. 95193