Distributed Communication via Global Buffer

by

David Gelernter and Arthur J. Bernstein
State University of New York at Stony Brook

Abstract. Design and implementation of an inter-address-space communication mechanism for the SBN network computer are described. SBN's basic communication primitives appear in context of a new distributed systems programming language strongly supported by the network communication kernel. A model in which all communication takes place via a distributed global buffer results in simplicity, generality and power in the communication primitives. Implementation issues raised by the requirements of the global buffer model are discussed in context of the SBN impementation effort.

1. Introduction.

Communication between processes executing in disjoint address spaces has long been a topic of interest to the designers of operating systems. This interest has increased with the development of high-level languages to support distributed programs. In this paper we discuss a proposal for one such language, Linda, which is being developed as an integral part of a project to build a network computer called the Stony Brook microcomputer Network (SBN). (By "network computer" we refer to a general-purpose MIMD machine for support of distributed programs.) Processes of a distributed program may be multiplexed on a single processor node or they may occupy separate network nodes. Formally the two cases are equivalent.

Research sponsored by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under Grant Number AFOSR 81-0197. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

In most inter-address-space communication models, communication involves copying from the address space of the sending process to the address space of the receiving process, or in some cases from the sender to a mutually-accessible port or "exchange"[1] from which the receiver reads in turn. We present, in the following, communication primitives based on another model, in which output and input statements are uncoupled and communicate through a (conceptually) infinite global buffer. We argue for the generality and usefulness of these primitives and discuss the SBN implementation.

SBN is a network computer of LSI-11 processor nodes, each with private memory, currently connected by word-parallel point-to-point buffered links. The network is configured in a torus--a square end-around grid in which each row and each column loops back on itself; each node has four nearest neighbors. Central to SBN's design is the integration of our distributed programming language with the network communication kernel. SBN is a "Linda machine" insofar as access to the hardware and the message-passing kernel is mediated by the language. Inter-address-space communication takes place in context not of system calls but of the structured programming language.

Linda is an intermediate-level language; its design emphasizes flexibility and efficient implementation on a network computer, and we intend to use it in constructing a distributed operating system. It contrasts most sharply with Argus[2], which is directed toward the building of consistent distributed data bases and as a result incorporates a number of features that are not relevant in our context. Linda has a no-wait output operation and a synchronizing input operation--thus placing it in the class, so far as synchronization goes, of such systems as PLITS[3], Guardian-extended CLU[4] and the Arachne network operating system[5]. This class is to be distinguished from remote-procedure-call languages such as DP[6]; inter-task communication in Ada[7] also follows the remote procedure call model. Languages such as SR[8] and *Mod[9] make up a third class that supports both forms of inter-address-space communication. Linda's idiosyncratic structured-memory model sets its apart, however, from all three classes in ways we discuss below.

Preliminary design of the language as a whole is discussed elsewhere[10]; Linda's basic communication primitives are the subject of the discussion following. Section 2 deals with design and section 3, implementation. Kernel routines are now operational on a 4-node prototype net. Section 4 presents conclusions. The SBN project in its entirety is discussed in [11].

2. The global buffer and its communication primitives.

2.1 Structured Memory

Linda programs run on a virtual structured memory machine (SMM) that implements structured memory (stm). Linda programs consist of a dynamically-varying set of simultaneously-active processes that communicate with each other via shared, structured memory, where structured memory is an abstraction encompassing both the global buffer and the communication primitives. An SMM defines a single structured memory that is accessible to all processes through Linda's communication operations. Structured memory has the following characteristics:

1. It may be described as an abstract type. It defines the three operations that Linda programs use for inter-process communication; it is accessible only via these three operations.

2. It may be implemented over a single physical memory or over many disjoint physical memories. Thus stm is physically distributable.

3. An stm is not an array of bytes but a collection of ordered tuples. Tuples may be added to, copied from or removed from the collection, but they may not be altered. Thus stm is atomic insofar as a tuple, to be altered, must be physically removed from stm and subsequently re-inserted with a new value. Each tuple is a physical object that is added to or removed from the collection only in its entirety, never in pieces; thus stm is multi-accessible insofar as any number of add-tuple, remove-tuple or read-tuple operations over stm may be executed simultaneously by parallel processes without damage to stored tuples or risk of inconsistent stm state. Stm is synchronizing insofar as a program seeking to remove or read a missing tuple suspends until the sought tuple has been added to stm.

4. Tuples stored in stm are accessed via name, not address. The physical addresses of stored tuples are not accessible to Linda programs. They are known only to the SMM itself, and they may change as the SMM moves tuples from place to place for convenience. Furthermore, since names may be structured (see below), stm is in fact not merely name-addressable but content-addressable.

2.2 Communication in structured memory.

2.2.1 Preliminary: the type "name".

Linda includes a distinguished type "name". Name-valued variables and constants are attached as identification tags to data inserted in stm, as discussed below.

2.2.2 in and out

The out statement appears as

$out(N,A_2,...A_j)$,

where $A_2...A_j$ is a list of actual parameters, and N is an actual parameter of type "name". Execution of the out statement results in the insertion of the tuple $N,A_2,...A_j$ into stm; the executing process continues immediately. The in statement appears as

$in(N,F_2,...F_k)$.

N must be an actual of type name. $F_2,...F_k$ may be either formals or actuals. If $F_2,...F_k$ are formals, then execution of the in statement results in reception of some currently-buffered type-consonant set of actuals whose first component is "N". When reception occurs, the received actuals are removed from stm and their values are assigned to the corresponding formals in the in statement. If no set of matching actuals is present in stm, in suspends until some set is available.

As noted, some or all of $F_2,...F_k$ may be actuals. Some or all of $A_2,...A_j$ in the out() statement's parameter list may furthermore be formals. These cases are discussed below.

Note that non-determinism is inherent in the definition of in and out, in the following sense. The statement in(N,...) may be executed simultaneously in several processes of a distributed program. Executing the statement out(N,...) under such circumstances makes a single copy of the output parameters available. Which in() statement receives them is logically non-determined. Similarly, when a single in() statement executes subsequent to many out() statements, which tuple the in() statement receives is non-determined.

If all but the first parameter in an in() or out() statement is omitted, then the null parameter list has been specified. The statement out(R) adds the "null set of actuals" labelled R to stm. The statement in(R) removes one instance of the null actual list labelled R from stm, then continues. If there are no instances of the R-labelled actuals list available when in(R) is executed, the executing process must wait until out(R) is executed. (Note that the semantics of in and out with null parameter list are identical to their semantics with non-null lists.) Thus an out statement with null parameter list, e.g. out(sem), is functionally equivalent to the semaphore operation V(sem). The statement in(sem) is equivalent to P(sem). A semaphore sem is initialized to value n by n repetitions of the statement out(sem).

2.2.3 The read-tuple operation in*

Let +t refer to any tuple added to stm by an out(+t) operation. Then -t designates the parameter list of any in statement that may legally receive tuple +t.

The in* statement appears as

```
in*(N,...F_k)
```

where N...F_k is a tuple precisely as for in(). When in*(-t) executes, if some tuple +t is currently stored in stm, a copy of it is received by in*(-t) and the process executing in* continues. The tuple +t itself, however, is not removed from stm. Thus in* is a read-tuple operation. If there is no instance of +t currently in stm, execution suspends until some instance is added, precisely as for in(). Together with the structured names discussed below, in*() allows the implementation of broadcast operations as described in section 2.4. (in*() is also central to dynamic process creation, which is discussed in [10] but is outside the scope of this paper.)

2.3 Structured names

As noted above, not all of the list $F_2 \ldots F_k$ cited by an in() or in*() statement need be formals. Any or all components on the list may in fact be actuals. The actual parameters appearing in a -t tuple (including the initial name-valued actual) collectively constitute a structured name. Thus the statement

```
in(P, i:integer, j:boolean),
```

requests a tuple with name "P". But it is also possible to write

```
in(P, 2, j:boolean);
```

in this case a tuple with structured name "P,2" is requested. The statements

```
in(P, i:integer, FALSE)
```

or

```
in(P, 2, FALSE)
```

are also possible. In the first case, the stuctured name is "P,,FALSE" and in the latter, "P,2,FALSE". All actual components of a -t list must be matched identically by the corresponding components of a +t tuple in order for tuple matching to occur. The in() or in*() statements with structured name may in other words be described as performing a relational-algebraic select operation over all stored tuples.

Just as components of -t may be actuals, any component of +t except the first (which is the mandatory name-valued actual) may be a formal. This case and the use of structured names generally are discussed in the examples below.

Two notes: (i) In the absence of a specific declaration by the programmer to the contrary, names that appear in program P are disjoint from names that appear in any other program. If P wishes to reference names created in some other program, P′s source must identify such names as "external". (ii) A formal declared in an in() or in*() statement′s parameter list is defined from the point at which the statement appears through the end of the immediately enclosing lexical block. Variables declared conventionally (rather than in the parameter lists of in() or in*() statements) may be used as formals by in() or in*() statements wherever they are lexically visible; in this context they must be prefixed by "var"--otherwise they will be interpreted as actuals and part of a structured name. Thus, in the sequence

```
i: integer; ... in(P, var i)
```

the integer variable i serves as an in-statement formal. In the sequence

```
i: integer; ... in(P,i),
```

i′s value is one component of the structured name P,i.

A final note: the current SBN/Linda implementation assumes that, if tuples added to stm by program P remain undelivered when P terminates, they may be deleted from stm. There is, however, no logical reason why stm may not be extended from RAM into backing store and used to store tuples that are meant to persist after P has terminated. In a system that implements long-term tuple storage, Linda programmers are called upon to specify explicitly the names of those tuples that are to persist in stm after program termination.

2.4 Examples.

Several properties of Linda are useful in providing the flexibility that is required of a systems programming language. Because in() and out() are uncoupled, one process can send to a second process, suspend itself by waiting for input and ultimately receive input from a third process. This property, together with the fact that name-valued constants and variables may be passed as parameters, makes it possible to program resource-allocation algorithms in a flexible way. The global buffer model allows out() and in() to be used not only for interprocess communication but for the creation and manipulation of variables that are global to disjoint address spaces. This in turn simplifies broadcasting and the development of systems involving multiple identical servers. Structured names allow programs to respond selectively to some subset of a general class of events. These characteristics are illustrated in the following examples.

Consider the design of a general resource server. A user requiring resource R sends its name, MY-NAME, as a parameter to the R-server, and then suspends itself by executing in(MY-NAME). When the resource is available the server responds with out(MY-NAME), thus allowing the user to resume and access the resource. When the access is complete, the user notifies the server by executing out(resource_free).

The resource server consists of two processes that execute concurrently in mutual exclusion, a feeder and a driver. Feeder and driver share access to a request queue. The feeder receives requests and inserts them into the queue. The driver waits for the resource to be free, then allocates it to the next waiting requestor. The general outlines of feeder and driver are

```
feeder =
  in(feeder, requestors_name: name);
  [
     if resource not busy
     then [out(requestors_name); busy := true]
     else insert requestors_name in QUEUE
  ]

driver =
  in(resource-free);

  [
     if QUEUE not empty
     then
      [ N := next requestors_name from QUEUE;
        out(N) ]
     else busy := false
  ]
```

Maintenance and manipulation of the service queue and continuation or further suspension of requestors is entirely in the hands of the server. The queue may be FIFO, device optimized, priority ordered, etc. as the server chooses. (An alternative is to provide the abstraction of a scheduled resource, in which use of the resource is provided by a nested call from the server. This simplifies the user interface and is appropriate in some situations.)

In the feeder-driver example, queues implemented as linked lists constructed out of conventional dynamic variables have been assumed. Conventional dynamic variables cannot be referenced from physically disjoint address spaces, and thus in the above example the only possible assumption is that feeder and driver are multiplexed on one network node. In fact, linked lists may be constructed in stm and accessed from disjoint address spaces; in this case, feeder and driver may run simultaneously on different processors and still share the request queue between them. In stm queues, names replace conventional pointers. A server that receives requests by executing

```
  in(R, r:req_type)
```

might construct a queue whose elements are tuples of type

```
  QR, n:name, r:req_type, nxt:name.
```

QR is a name prefixed to all elements of R´s queue, n is the name of a particular tuple and nxt is the name of the tuple following n in the queue.

The use of a name in stm as a rendezvous point for communication is a departure from techniques employed in other distributed languages. In other proposals a process must generally name its corespondent, a port attached to its correspondent or a port declared in a module within which the correspondent executes. The last scheme (as represented, for example, by Guardian-extended CLU[4]) is most general, since it allows messages directed to the port to be received by any process executing within a given module. Thus a server module can be programmed containing n identical server processes (n is an implementation parameter hidden from users), and service requests may be accepted by any of these processes. Since all processes execute within one module they must, however, all execute on one processor node. Linda carries the generality of this scheme one step further in allowing servers on several nodes all to input data from the same rendezvous point. For example, an idle node in the network might schedule an "idle" process that executes

```
  in(idle, j: job_description).
```

A process wishing to fork a new process k communicates with any such idle process by executing out(idle, k). Note that in this example, the number and location of processes willing to accept information addressed to the global name "idle" varies dynamically. The issue of providing a rendezvous with the nearest idle process may be dealt with in the kernel.

An extension of the multiple server example illustrates the use of structured names. Consider the case of a user who must engage in a conversation with some server involving a sequence of requests. It may be immaterial which of several simultaneously-active identical servers answers the user´s initial service request. The same server that handles the initial request must, however, handle all of that user´s subsequent requests because each response depends upon the current state of the conversation. (This set of requirements is called "conversational continuity" in [4]). It is furthermore desirable that the user not be aware of the presence of multiple servers; the identity of a particular server is an implementation detail and should not be part of the user´s view of the resource. Assuming that conversations always start with transmission of the distinguished start-up request "start", a conversation from a user´s point of view takes the form:

```
out(S, start, U, request_description);
in(U, response: response_type);
{ out(S, request_type, U, request_description);
  in(U, response: response_type)}*
```

where S is the name used by all servers providing the service, U is the user´s unique name and the asterisk means 0-or-more repetitions. Idle servers--those not currently conversing with a user--receive start-up requests by executing

```
in(S, start, N: name, r: request_descriptor).
```

Subsequently, for as long as a given conversation continues, servers receive requests by executing

```
in(S, d: request_type, N, r: request_descriptor).
```

Structured names and in*() together allow the implementation of broadcast via stm. Suppose some process P wishes to broadcast a series of messages to some set of processes Q1, Q2...Qn. P tags all of its broadcast messages with some name "B", and numbers them sequentially. Thus P´s first broadcast takes the form out(B, cntr, m), where m is the message and cntr is set initially to 1. P increments cntr before each subsequent broadcast. Each of the Qi receives broadcasts by executing in*(B, cntr, m: message_type), with cntr again initially set equal to 1 and incremented before each subsequent in*() operation.

Structured names allow processes to monitor some selected subclass of events. Suppose in the above example that an exception handler E is to be added to the program; E is to react when the value of cntr reaches max_val. E may be written

```
  in*(B, max_val, m: message_type);
  do something about overflow.
```

In an obvious generalization of the naming rules described thus far, formals may be allowed in out-statement lists precisely as actuals may be allowed in in-statement lists--provided the first element in an out-list is always a name-valued actual. A buffered tuple containing formals may be received by in-statements whose parameter lists contain a type-consonant actual wherever the out-list has a formal. (A formal in an out-list may be matched only be an actual, never by another formal.) Suppose, for example, that network nodes in quiescent state execute

```
  in(idle, i, j:job_description),
```

where i is an integer that uniquely identifies each node. A process with a pending job k that must be executed on node 6 accordingly executes

```
  out(idle, 6, k).
```

If, on the other hand, k may equally well be executed on any idle node, the appropriate statement is

```
  out(idle, n:integer, k),
```

which adds to stm a tuple that may be received by an in(idle, i, ...) statement for any value of i.

Supporting Linda's operators and structuring tools will require some sophistication and complexity both in compiler-generated code and in the runtime kernel. Compiler-generated code and the kernel together define the structured-memory machine upon which Linda programs execute. We discuss the SMM below.

3. The SBN Structured Memory Machine.

Linda programs execute on a virtual SMM whose functions are implemented by compiler-generated code and the SBN kernel in cooperation. The basic function of the SMM is to accept parameters from out() statements and deliver them to in() statements. Since out() and in() may execute on separate nodes, the SMM must implement inter-node communication. Since they may execute on the same node, and since furthermore many in() and out() statements may execute concurrently on one node, the SMM must implement intra-node communication and short-term processor scheduling as well.

The SMM in performing these functions must incorporate a number of special features. It must implement a global name space; it must provide buffering for tuples added to stm. Buffer space in stm is conceptually infinite. The infinite buffer is simulated by means of a flexible buffer space that expands as messages are added and contracts as they are removed.

We discuss the global names space in section 3.1, buffering in 3.2 and the SBN implementation in 3.3.

3.1 The Global Name Space.

A name space that is global to all modules of a distributed program makes it impossible to resolve global-name references statically before runtime. Even were the n modules of a distributed program assigned statically to some n modules of the network computer, a global name may be referenced at runtime on any or all of these n nodes. Some method must therefore exist for matching +t tuples to in(-t) statements at runtime. Once such a method has been provided it is furthermore unnecessary, so far as name resolution goes, to assign modules of a distributed program statically to network nodes. Allowing the network computer's operating system maximal freedom to locate and re-locate distributed jobs dynamically within the network is in fact an important goal in itself.

Runtime resolution of name references may be handled in either of two general ways. Name-node mapping information may be concentrated in central directory nodes, or it may be distributed network-wide, for example by means of a network-wide broadcast.

The maintenance of a central directory leads to congestion and vulnerability and is inappropriate as a kernel function. Ideally the kernel is a small, simple program basically identical on every node. Distinguishing certain kernels as directory servers, and requiring all others to submit service requests to these, involves the kernel in an undesirable degree of complexity and specialization. If the kernel, then, is to implement the global name space--and it must, because Linda is to be supported directly by the kernel--then distributed storage of name-node mapping information is an attractive approach.

The combined weight of these design decisions is to make the mapping of logical names to physical addresses a function of the (virtual) machine itself. Conceptually, program modules and the names they define may be mapped to nodes and moved between nodes as freely as pages may be moved among page frames in a paged memory. To support this flexibility, the paged machine translates virtual to physical addresses in the hardware on a per-reference basis. The analogous function on SBN is the translation of global names to network addresses on a per-reference basis by the communication kernel. Against the potential benefits of this flexible architecture must be balanced the cost and complexity of implementing internode communication and runtime resolution of name references. The power of the SBN design will be realized only if the communication kernel can be implemented efficiently. We address specifically the issue of efficient runtime name resolution in the remainder of the section.

We define in [12] a method of network-state storage called "uniformly distributed", where "network state" is the collection of all data descriptive of any particular node that may be of interest to the network generally. The fact that a tuple with name M is currently required by an in() statement on node i is an instance of a state datum.

Informally, a network state storage scheme is uniformly distributed if no node stores more of the current state than any other node. Uniformly distributed state storage schemes fall into a discrete spectrum. At one end of the spectrum, state data is broadcast to every node, and each node maintains its own complete copy of the state of the entire system. At the other end, state information remains local to the node on which it originates. In this case, to discover the current state, a given node must broadcast a query to the entire network; nodes respond to this broadcast query with information extracted from their local states.

If state data is read much more frequently than written, then clearly the first scheme is more efficient in terms of total communication and processor bandwidth expended by all reads and writes. In this scheme, writing is expensive (a broadcast is required), but reading is cheap. If writes are much more frequent than reads, then the second scheme will be more efficient. If, on the other hand, reads and writes occur with roughly equal frequency, then a third scheme, intermediate between the first two, is more efficient than either. ("Roughly equal" has a precise definition: on an N-node net, roughly equal means equal within a factor of $N^{1/2}-1$ [12].) In the intermediate scheme, assuming an N-node network, writing the network state consists of broadcasting state data to a pre-arranged set of $N^{1/2}$ nodes called the write set. Reading the state consists of broadcasting a query to a different set of $N^{1/2}$ nodes called the read set. If the read set is chosen correctly, its nodes will store among them the entire network state. This condition is assured if each node's read set has a non-null intersection with each other node's write set.

Consider now the problem of implementing in and out. By definition, the operation write(-t,j) will add to the network state the datum "-t is available on node j", in other words that in(-t) has been executed on node j and that some instance of +t is accordingly desired. The operation read(-t) will consult the network state and return the identity of some node on which in(-t) has been executed and on which an instance of +t is accordingly needed. If there is no such node, the read(-t) request will remain outstanding until there is such a node and will then return that node's identity. (Note that "read" and "write" are internal to the communication kernel. They are not part of Linda.). Abstractly, then, node i's kernel may implement an out(+t) operation as follows:

```
j := read(-t);
send +t to node j
```

in(-t) as executed on node j is implemented

```
write(-t,j);
await receipt of +t.
```

Note that the implementation of in() and out() is not atomic. As a result, two instances of +t may be sent to the same node j, in which case the second must be re-inserted into the global buffer.

Let "-t-state" refer to all data in write sets pertaining to the whereabouts of -t-tuples (i.e., of in(-t) statements) in the network. If in() and out() are executed with roughly equal frequency over all Linda programs--this seems to be a reasonable assumption, although there are complicating factors that we discusss below--then the network's -t-state will be read and written with roughly equal frequency. Given that the -t-state is to be stored in uniformly-distributed fashion, write(-t,i) and read (-t) are accordingly best implemented using the intermediate $N^{1/2}$-node broadcast technique.

An implementation of the intermediate scheme on the SBN torus works as follows. For each node i, i's write set is its row and i's read set is its column. When in(P...) executes on node j, j's kernel executes write(-P,j) by sending to each of the $N^{1/2}$ nodes in its row a message reading "a tuple labelled 'P' is needed on node j". This process is referred to as "constructing a P-in-thread". (Note that we deal at this point only with simple names. Structured names are discussed below.) When out(P,...) executes on node i, i's kernel sends to each of the $N^{1/2}$ nodes in its column a message reading "a tuple labelled 'P' is available on node i"; this process is called "constructing a P-out-thread". When a P-in-thread either intercepts or is intercepted by a P-out-thread on node k, a notification message is generated on node k and sent to node i instructing i to send tuple (P...) to node j. When and if the tuple is delivered, accepted and acknowledged, both P-in-thread and P-out-thread are removed. Some other (P...) tuple may, on the other hand, arrive at node j before node i's does. If so, node i's tuple is not accepted and i's P-out-thread remains in place.

Note that an out-thread registers data in all nodes of a read set just as an in-thread does in a write set. But the data recorded in an out-thread is treated as a persistent state query, not as part of the network state.

The overhead of constructing in- and out-threads may be substantially reduced in the following ways. First, once node j has determined that tuple (P...) is needed on node i, it associates "P" with "i" in a name-address cache and sends all subsequent (P...) tuples directly to i, bypassing out-thread construction, until i rejects one such tuple. This scheme is exactly analogous to the runtime conversion of logical to physical addresses in a virtual memory system. A related factor decreases in-thread construction. Linda allows the programming of procedure-like blocks headed by in() statements to which execution returns repeatedly. in-threads associated with such in-statements needn´t be repeatedly torn-down and reconstructed; they may be left in place for as long as the associated procedure block remains invocable.

in*() statements are implemented precisely as in() statements are, so far as in-threads go. But regarding the associated out-thread operations there are two possibilities. Suppose in*(-t) executes on node i and out(+t) has been executed on node j. Execution of in*(-t) might cause a copy of +t to be transported to i, leaving the original undisturbed on j. On the other hand, the original might itself be transported to i, in which case subsequent in*(-t)´s will find it there--not, in other words, on its node-of-origin i, but on its node-of-last-reference j. In the former case the +t-out-thread remains undisturbed with its origin on i. In the latter, i´s +t-out-thread is torn down and j must construct a new one with itself as origin. Clearly the second scheme involves greater thread-construction overhead than the first. The second scheme on the other hand has what might be the desirable property of distributing tuples to the network sites where they are needed, rather than leaving them forever at their creation sites. Note that the first scheme has the effect of unbalancing reads and writes over the -t-state by increasing in-thread construction (writes) relative to out-thread construction (reads). If measured performance in a given system indicates a significant imbalance of read-state as opposed to write-state operations, the size of read and write sets may be adjusted as described in [12] to maintain good performance with uniform state distribution. in*() has not yet been implemented on SBN and no choice of strategies has been made.

Threads as described above and currently implemented perform only simple-name matching. They ensure that a tuple is delivered to an in() statement whose first component matches the tuple´s. But that in() statement may in fact specify a structured name, and if the arriving tuple fails to match all fields of the structured name identically, it will be rejected and must be re-transmitted to another candidate in() statement. A more efficient implementation involves the propagation not just of names but of structured names in in-threads. A structured name is propagated as a list of fields that a tuple must match in order to be acceptable to a given in() statement; out-threads continue to propagate simple names only. Notification messages generated when in- and out-threads intersect contain a copy of the structured-name list associated with the in() thread, and no tuple that fails to match a structured name is transmitted.

Regardless of topology, as the number of nodes in a network grows, the problem of increasingly long message path lengths becomes acute, contributing to both delay and processing overhead. A mechanism is then required for finding a node within a certain neighborhood that can provide a particular service (e.g. a nearby idle node). It might even be the case that information concerning the state of nodes whose distance from node i exceeds some threshold is irrelevant to i. The issue of locality is related to another problem. The size of the stm, and thus the amount of information stored on each node, grows with the size of the network. A technique for limiting the latter will be required on large nets.

One solution to these problems lies in a hierarchical approach. Read and write sets confined to a neighborhood of i are defined and referred to as first-order sets. The neighborhood is a set of nodes whose distance from i does not exceed some fixed limit. Nodes within the neighborhood communicate through their first-order sets. On the torus, i´s first order sets might be all nodes in its row or column that are no more than k distant from it. It is now no longer the case that a read set contains one node from the write set of each other node. Consulting a first-order read set only yields information about a neighborhood, and this is insufficient if a node is interested in the state of a larger area. It does, however, reduce the amount of data stored on each node as well as encourage short communication paths.

In order to reach a larger area, nodes performing a second order function are introduced. A subset of network nodes is selected to perform this function. These nodes are regularly spaced and chosen so that the neighborhood of each node i contains one such node. Information concerning i´s state which is to be globally distributed is not only broadcast to i´s first-order write set, but picked up by the second-order node, s, in i´s neighborhood. Node s then broadcasts this information to its second-order write set, which is composed only of second-order nodes, each of which lies outside of i´s neighborhood. Similarly, s has a second-order read set composed of only second-order nodes.

Clearly a node that performs a second-order function stores considerably more information than nodes that do not, since it must perform a first-order function as well. Such nodes, however, are sparse in the net. Second-order nodes must be distributed in such a way that some node within the first-order write set of each node is included within the first-order read set of some second-order node.

As a final comment on large nets, note that the techniques described above are not restricted in applicability to two-dimensional grids. The hypercube might well be a more appropriate topology for a large net. Node i ´s intermediate read and write sets on an n-dimensional hypercube are defined, for n even, as two n/2-dimensional hypercubes that intersect at i ([12]).

3.2 Buffering.

The SMM is required to simulate Linda´s infinite global buffer. System buffer space must expand as needed; buffer space (as required by the orthogonality property) must be provided for undelivered tuples. These requirements are handled by making the allocation of buffer space largely the function of compiler-generated code.

Allocation of buffers for arriving tuples is in principle a simple extension of the allocation of activation frames in the implementation of sequential languages. When in(P...) executes, a buffer for the actuals is allocated in the execution stack of the process executing in(); this buffer is referred to as buffer P. An arriving tuple is stored in buffer P after match and delivery have occured. Outbound tuples are stored in per-process heaps maintained by compiler-generated code until they are deleted from stm. Header information required by the communication system--dealing with the length of the tuple, its logical address and other basic control characteristics--is prefixed to the buffered outgoing tuple by compiler-generated code.

If a process terminates with undelivered tuples, responsibility for the buffers in which they are stored devolves on the kernel and these buffers are not deallocated until their contents have been delivered. (When all processes of a given distributed user job have terminated, however, that job's undelivered tuples may be erased--until an implementation of long-term tuple storage extends stm onto backing store.)

Abstractly, out() never causes the executing process to suspend execution. For implementation reasons, out(N...) will in fact sometimes cause execution to be suspended. If no more storage space is available to the process executing out(N...), it suspends until more space is available. This property is required on grounds of practicality, but it is irrelevant to the logic of out(). At no point are synchronization constraints applied to out(). At no point is reception of the actuals output by out() guaranteed by the fact of execution's continuing past the out() statement.

3.3 Implementation.

SBN's kernel consists of a hop-level kernel (HLK) and a message-level kernel (MLK). The HLK is subdivided into a link-sublevel and a packet-sublevel. The HLK link-sublevel includes drivers for the communication lines, hop-level packet-exchange protocols and next-hop routing. The packet-sublevel implements packet buffers, output queues and the packet deadlock prevention algorithm described in [13]. The MLK handles end-to-end message-exchange protocol, intranode message exchange and short-term processor scheduling.

in-threads and out-threads are stored in thread tables within the HLK on each node. Ideally, all in-thread tables on a row and out-thread tables in a column contain identical information (allowing for the normal latency of thread-propagation messages). In fact, node failures, delays and table overflows make it likely that this will not always be the case. The problem of inconsistent thread tables may be rectified by requiring the node which is the source of an in- or out-thread to rebroadcast it periodically. The basic matching scheme will then work correctly despite temporary anomalies. The rebroadcasting scheme furthermore allows table cleanup to be performed simply by deleting entries that are not refreshed within some interval.

4. Conclusions.

The SBN/Linda design introduces or incorporates a number of unusual features. On the one hand, the simple and general abstraction of a structured memory referenced via in(), in*() and out(); on the other, threads as an orthogonal global name space implementation technique. But the system's fundamental and most important characteristic is its integrated design, particularly the relation of the language to a virtual machine and network architecture designed specifically to support it.

The discussion has emphasized ideas and techniques rather than engineering specifics. Indeed, fundamentals of SBN hardware may be drastically revised in the near future. The incorporation of front-end processors is being studied and the need for upgraded communication hardware is obvious. (The relationship between thread construction and bussed architectures makes the use of busses attractive.) Our hope is that such changes will have no major effect on our language and system-architecture ideas or on the implementation techniques we have discussed.

Our ideas will be tested in practice as the implementation effort proceeds.

Acknowledgements. Thanks to Mauricio Arango for major contributions to the implementation effort, and to Suresh Jagannathan for his work on the MLK.

References.

1. W.D. Sincoskie and D.J. Farber, "The Series/1 distributed operating system: description and comments," in Proc. Fall Compcon 80, Washington DC, Sept. 1980 p. 579.

2. B. Liskov and R. Scheifler, "Guardians and actions: linguistic support for robust, distributed programs," MIT Laboratory for Computer Science, Computation structures group memo 210, Nov. 1981.

3. J.A. Feldman, "High level programming for distributed computing," Comm. of the ACM, 22,6, June 1979 pp.353-368

4. B. Liskov, "Primitives for distributed computing," in Proc. 7th Symp. on Operating System Principles, Pacific Grove, Ca. Dec. 1979 p. 33

5. M. Solomon and R. Finkel, "The Roscoe operating system," in Proc. 7th Symp. on Operating System Principles, Pacific Grove, Ca. Dec. 1979 p. 108

6. P. Brinch Hansen, "Distributed Processes: a concurrent programming concept," Comm. of the ACM, 21,11, Nov. 1978 p. 934

7. "Reference Manual for the Ada Programming Language," United States Department of Defense, July 1980

8. G.R. Andrews, "Synchronizing Resources," ACM Trans. on Programming Languages and Systems, 3,4 Oct. 1981 p.405

9. R.P. Cook, "*MOD--a language for distributed programming," IEEE Trans. on Software Engineering, SE-6,11 Nov. 1980 p.563

10. D. Gelernter, "Principles of a Medium-Level Language for Distributed Systems Programming," tech. report no. 81-023, Department of Computer Science, SUNY at Stony Brook, Aug. 1981.

11. D. Gelernter, "An Integrated Microcomputer Network for Experiments in Distributed Programming," tech. report no. 81-024, Department of Computer Science, SUNY at Stony Brook, May 1981 rev. Dec. 1981.

12. D. Gelernter, A.J. Bernstein, "Global Name Spaces on Lattice-Connected Network Computers," tech. report no. 80-011, Department of Computer Science, SUNY at Stony Brook, Oct. 1980 rev. Oct. 1981.

13. D. Gelernter, "A DAG-based algorithm for prevention of store-and-forward deadlock in packet networks," IEEE Trans. on Computers, C-30,10 Oct. 1981 p. 709