To be presented at the Workshop on Future Directions in Network Architecture (FDNA'03),
An ACM SIGCOMM 2003 Workshop, August 27, 2003, Karlsruhe, Germany.

An End-to-End Approach to Globally Scalable Programmable Networking

Micah Beck Terry Moore James S. Plank

Logistical Computing and Internetworking Laboratory
Computer Science Department
University of Tennessee
1 865 974 3548

{mbeck, tmoore, plank}@cs.utk.edu

ABSTRACT

The three fundamental resources underlying Information Technology are bandwidth, storage, and computation. The goal of wide area infrastructure is to provision these resources to enable applications within a community. The end-to-end principles provide a scalable approach to the architecture of the shared services on which these applications depend. As a prime example, IP and the Internet resulted from the application of these principles to bandwidth resources. A similar application to storage resources produced the Internet Backplane Protocol and Logistical Networking, which implements a scalable approach to wide area network storage. In this paper, we discuss the use of this paradigm for the design of a scalable service for wide area computation, or programmable networking. While it has usually been assumed that providing computational services in the network will violate the end-to-end principles, we show that this assumption does not hold. We illustrate the point by describing Logistical Network Computing, an extension to Logistical Networking that supports limited computation at intermediate nodes.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design — *distributed networks, network communications, store and forward networks*

General Terms

Design

This work is supported by the National Science Foundation (NSF) Next Generation Software Program under grant # EIA-9975015, the Department of Energy Scientific Discovery through Advanced Computing Program under grant # DE-FC02-01ER25465, the NSF Internet Technologies Program under grant # ANI-9980203, and the Center for Information Technology Research. The infrastructure used in this work was supported by the NSF CISE Research Infrastructure program, EIA-9972889 and Research Resources program EIA-0224441.

Keywords

Logistical Network Computing, distributed state management, active networking, programmable networking, store and forward network, asynchronous communications, network storage, end-to-end design, scalability, Internet Backplane Protocol.

1. INTRODUCTION

The mounting wave of enthusiasm for the idea of building a new "cyberinfrastructure" draws much of its inspiration from the manifest success of the Internet as a catalyst for collaboration within the research community worldwide [1]. Various well-publicized visions of the technology-rich environments of the future — distributed collaboratories, global computing grids, ubiquitous computing, pervasive computing, autonomic computing — seek to expand and amplify the Internet's power in this regard. But when looked at through the lens of the Internet's success, even a casual review of such visions makes two points about such a next generation infrastructure abundantly clear: First, it must be more general than the current Internet, delivering a much more capable set of services and resources. Second, and less well remarked, it must scale at least as well as the current Internet does, and better along some dimensions Yet for infrastructure builders dealing with information technology in the wide area, i.e. for the Networking and Distributed Systems communities, this required combination of generality and scalability has proved to be extremely elusive.

On one hand, Networking has traditionally been concerned with "data communications," or the accurate and timely transfer of bits from one host system to another. This limited scope has allowed it to apply stringent rules of architectural discipline, notably the end-to-end principles [2], to the problem of designing systems that can scale. The obvious example is the Internet, which has successfully scaled to reach every country on the globe, and may soon reach most communities. However, with the recent exception of Logistical Networking [3], attempts to generalize the network to include storage or compute services, such as Active Networking [4], have generally run afoul of the end-to-end principles [5] and proved difficult to scale up.

On the other hand, while the designers of distributed systems would like to achieve Internet-levels of scalability, Distributed Systems research grew out of work on operating systems, and its scope has always been broader. Since its goals require more general services, including storage and computation, the design of these systems almost never invokes the end-to-end principles. Niches that have developed with the goal of enabling new functionality in the wide area, such as storage networking and "Grids," routinely treat scalability as a secondary matter, to be dealt with after new protocols and intermediate nodes have been designed and tested.

Our position is that a successful way to create a scalable, programmable network, capable of providing the kind of generality that many advanced applications and environments will require, is to integrate into the network a compute service that adheres to the end-to-end principles.

Given previous experience and prevailing attitudes, the most obvious and serious objection to this position is that it advocates doing something that cannot be done, but we argue here that it can be. Since our proposed approach leverages our previous work in the area of *Logistical Networking (LoN)*[6], we call it *Logistical Network Computing (LoNC).* Since the analysis we give here extends the analysis of globally scalable network storage that we previously presented at SIGCOMM 2002, interested readers are directed to that work [3].

2. Building on Logistical Networking

For several years we have been exploring the hypothesis that the Internet design paradigm for scalable resource sharing, which is based on the end-to-end principles [2, 5], can be applied more generally to both storage and processor resources. Our initial focus has been on storage, and under the heading of *Logistical Networking (LoN)*, we have used the Internet model to create a highly scalable network storage service based on the *Internet Backplane Protocol* (*IBP*) [7]. Because it can scale globally, IBP supports the incorporation of shared, interoperable storage elements, called IBP "depots," into the network, creating a communication infrastructure that is enriched with storage which is exposed to application developers and end users in an unprecedented way.

Since computation is inherently more complex than storage in various ways, it is natural to assume that LoNC will be inherently more challenging than LoN. This is certainly true. Yet there are two good reasons to set the stage for the discussion of LoNC by briefly reviewing a few main points about LoN, whose design and implementation preceded it.

The first reason is that it is easier to see the outlines of the basic idea in the context of network storage than it is in the context of network computation. Although the same end-to-end paradigm is being applied in both cases, the relative simplicity of storage, and its obvious parallels with the case of bandwidth, show more clearly how the model can be extended to a new resource.

Second, and more importantly, the work on LoNC builds directly on the storage service that LoN supplies, and this provides the logistical approach to computation in the network with a tremendous advantage that previous efforts in this general area did not possess: *a scalable, general purpose solution to the problem of managing the state of distributed applications*. The *absence* of such an infrastructure for interoperable state management is well known to present a major impediment to the creation of advanced distributed applications that can actually be deployed [8]. Since LoN technology provides a generic storage service, which is exposed for application scheduling and is scalable to the wide area, it lays the foundation for solving this problem and thereby opens up opportunities for new lines of attack on the problem of scalable network computing. The basic outlines of LoN, therefore, are essential context for understanding Logistical Network Computing.

As the name suggests, the goal of Logistical Networking is to bring data transmission and storage within one framework, much as military or industrial logistics treat transportation lines and storage depots as coordinate elements of one infrastructure. Achieving this goal requires a form of network storage that is globally scalable, meaning that the storage systems attached to the global data transmission network can be accessed and utilized from arbitrary endpoints. To create a shared infrastructure that exposes network storage for general use in this way, we set out to define a new *storage stack* (Figure 1), analogous to the Internet stack, using a bottom-up and layered design approach that adheres to the end-to-end principles. We discuss this design model in more detail below, but the important thing to note here is that the key to attaining scalability using this model lies in defining the right basic abstraction of the physical resource to be shared at a low level of the stack. In the case of storage the Internet Backplane Protocol (IBP) plays this role.

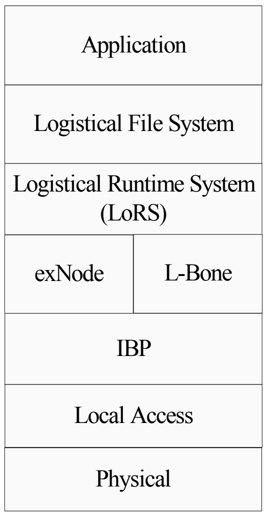

Figure 1: The Network Storage Stack

IBP is the lowest layer of the storage stack that is globally accessible from the network. Its design is modeled on the design of IP datagram delivery. Just as IP is a more abstract service based on link-layer datagram delivery, so IBP is a more abstract service based on blocks of data (on disk, memory, tape or other media) that are managed as "byte arrays." By masking the details of the storage at the local level — fixed block size, differing failure modes, local addressing schemes — this byte array abstraction allows a uniform IBP model to be applied to storage resources generally. The use of IP networking to access IBP storage resources creates a globally accessible storage service.

As the case of IP shows, however, in order to scale globally the service guarantees that IBP offers must be weakened, i.e. it must present a "best effort" storage service. First and foremost, this means that, by default, IBP storage allocations are time limited. When the lease on an IBP allocation expires, the storage resource can be reused and all data structures associated with it can be deleted. Additionally an IBP allocation can be refused by a storage resource in response to over-allocation, much as routers can drop packets; such "admission decisions" can be based on both size and duration. Forcing time limits puts transience into storage allocation, giving it some of the fluidity of datagram delivery; more importantly, it makes network storage far more sharable, and easier to scale.

The semantics of IBP storage allocation also assume that an IBP storage resource can be transiently unavailable. Since the user of remote storage resources depends on so many uncontrolled, remote variables, it may be necessary to assume that storage can be permanently lost. Thus, IBP is a "best effort" storage service. To encourage the sharing of idle resources, IBP even supports "soft" storage allocation semantics, where allocated storage can be revoked at any time. In all cases such weak semantics mean that the level of service must be characterized statistically.

IBP storage resources are managed by "depots," which are servers on which clients perform remote storage operations. IBP client calls fall into three different groups [7]: **`IBP_allocate`** and **`IBP_manage`** for storage management; IBP_store, IBP_load, **`IBP_copy`**, and **`IBP_mcopy`** for data transfer; and **`IBP_status`**, for depot management. The **`IBP_allocate`** function is the most important operation. It is used to allocate a byte array at an IBP depot, specifying the size, duration (permanent or time limited) and other attributes. A chief design feature is the use of capabilities (cryptographically secure passwords) [9]. A successful **`IBP_allocate`** call returns a set of three capabilities for the allocated byte array — one for reading, one for writing, and one for management — that may be passed from client to client, requiring no registration from or notification to the depot

The other component of the storage stack that is critical to the logistical approach to network computation is the *exNode* [10]. Building on end-to-end principles means that storage services with strong properties — reliability, fast access, unbounded allocation, unbounded duration, etc.—must be created in higher layers that aggregate more primitive IBP byte-arrays beneath them. To apply the principle of aggregation to exposed storage services, however, it is necessary to maintain state that represents such an aggregation of storage allocations (e.g. distribution or replication across multiple depots), just as sequence numbers and timers are maintained to keep track of the state of a TCP session.

Fortunately there is a traditional, well-understood model to follow in representing the state of aggregate storage allocations. In the Unix file system, the data structure used to implement aggregation of underlying disk blocks is the inode (intermediate node). Under Unix, a file is implemented as a tree of disk blocks with data blocks at the leaves. The intermediate nodes of this tree are the inodes, which are themselves stored on disk.

Following the example of the Unix inode, a single, generalized data structure called an external node, or exNode, has been implemented to aggregate byte arrays in IBP depots to form a kind of pseudo-file (Figure 2). We call it a pseudo-file because it lacks features (e.g. a name, ownership metadata, etc.) that a file is typically expected to have. To maximize application independence and interoperability, exNodes encode their resource information (e.g. IBP capabilities, URLs, etc.) and associated metadata in XML.

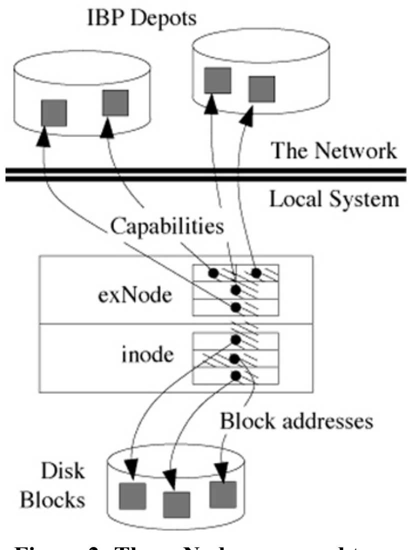

Figure 2: The exNode compared to a Unix inode.

The exNode provides the basis for a set of generic tools, called the *Logistical Runtime System* (LoRS), for implementing files and other storage abstractions. The LoRS tools, which are freely available at (http://loci.cs.utk.edu), can be used in a wide area testbed of IBP depots (currently +15TB) called the *Logistical Backbone* (*L-Bone*). Today the L-Bone encompasses more than 200 public nodes in 19 countries, and this deployment of LoN technology provides a rich platform for experimentation on a new approach to scalable network computation.

3. SCALABILITY OF NETWORK COMPUTATION

In designing a scalable network computing service, we follow a methodology that is simple, but which reverses the typical order of thought in the design of distributed systems. Rather than starting with an idea of what level of functionality we require of the network, or what sort of intermediate nodes we want to build, we start with the requirement that the system scale, using adherence to the end-to-end principles as the means by which such scalability can be achieved. This demand is stringent enough to dictate many important features of the basic service.

The most important consequence of *requiring* scalability is that the semantics of the service must be simple and weak [5]. If the semantics are too complex, it will fail the requirement that services implemented at intermediate nodes be generic. Likewise, if the guarantees made by the service are too strong, then it will not compose with a scalable communication network, such as the Internet, without breaking. This approach was followed in the design of IP for wide area communication, or Internetworking, and in the design of the Internet Backplane Protocol (IBP) for wide area storage services, or Logistical Networking [3]. In this section, we will show how a wide area network compute service can be created by systematically applying this same approach to the basic semantics of network computation. In analyzing different dimensions of the service — availability, continuity, statefulness, correctness, and security — the way in which corresponding aspects of IP and IBP were designed will serve as guides. The resulting design parameters will be used to specify a computational extension to IBP, thereby defining LoNC.

3.1 Availability

The core service of the Internet is best-effort forwarding of IP datagrams at an intermediate node. The best-effort nature of the service means that any particular router may be unavailable. Internet routing takes advantage of redundancy in the underlying topology of the network to find different paths between endpoints that avoid the unavailable router. This strategy is good enough to ensure a datagram delivery service between endpoints that is acceptably reliable.

In LoN, the core service is storage of data on, and transfers of data between, intermediate nodes that implement IBP. Like datagram service for the Internet, IBP is a best-effort service, so users must be prepared for an intermediate node to be unavailable for unknown periods of time. But unlike communicating applications, which maintain data state at the endpoint in anticipation of a possible loss of connectivity, storage applications must rely on redundancy across multiple intermediate nodes to overcome outages. As a result, although applications cannot rely on predictable locality in their access to storage, the service seen by end-users is acceptably reliable.

In the case of LoNC, the core service will be the transformation of stored state. As with IP and IBP, a best-effort computational service has to allow for the unavailability of an intermediate node that performs computation. In order for the service to proceed when a particular node becomes unavailable, redundant resources must be used to perform the computation on some other node. This means that the data on which the computation acts must be accessible even in the presence of failures in nodes that may both store and compute; such state-management strategies fall into the domain of LoN. Although the need to find alternate compute nodes and retrieve stored data from remote sources means that applications cannot rely on predictable locality in their access to computation, the service seen by end-users will be acceptably reliable.

In a scalable Wide Area Network (WAN), computing resources can be intermittently unavailable (or available only with inadequate quality of service) due to a number of conditions in the network, including traffic congestion, routing problems, topology changes and malicious interference. These conditions are resolved in time frames ranging from less than a second to longer than days. In the face of such problems, a variety of familiar end-to-end strategies should exist for ensuring availability. These range from simple retry, to redundant computations spread across the network, to maintaining high-latency, high-reliability systems. Obviously, these approaches must be implemented at the end-points, both in order to ensure delivery to the end-point and to achieve the necessary level of sensitivity to the requirements of the end-point operating systems, applications and users.

3.2 Fragmentation

The basic difference between circuit switching and packet networking is the unit of service: once a circuit is set up, the amount of data that can flow through it is in principle unbounded, whereas every intermediate node in a packet network has a maximum transmission unit (MTU). End-to-end services such as TCP streams that are unbounded in their handling of data are implemented by aggregating the underlying datagrams, which themselves may be implemented by aggregating underlying packets.

As applied to LoN, restricting the basic unit of service means that both the size *and the duration* of storage allocation have limits that are imposed by the *storage intermediate node* or *depot*. Limiting the size of allocations means that a large data object may have to be spread across multiple depots. This type of fragmentation based on size limitations is well known in all storage systems; but fragmentation due to temporal restrictions is not. Limiting the duration of a storage allocation means that every allocation is a lease, and while it may be renewable, this is subject to availability of resources and not guaranteed. In the face of an expiring allocation that cannot be renewed, an endpoint must be prepared to transfer the data to another depot or to lose it. This is a highly unusual attribute for a storage service, and is one of the key points separating LoN as a scalable service from other storage services.

When applied to computation, fragmentation of service implies that the size and duration of a computational service have limits that are imposed by the *computational intermediate node*. (As we explain below, in LoNC, this intermediate node is a modified IBP depot.) The limit on size means that the local state that any one computational service transforms has a limit. Such limits on process size exist in all operating systems. But as with storage, the limit on duration of a computation is less intuitive, although it has historical roots.

Fragmenting a computation means that any particular request for service has a fixed upper bound on the computational resources that it can consume. As with storage, this request may be renewable, but the endpoint must be prepared for the case in which it is not renewed due to resource oversubscription. When a computational fragment is done, the data it has just transformed will, in general, still be available. Continuing the computation means making a new service request on that or another computational intermediate node.

However, in all these cases — bandwidth, storage, and computation — the reason for limiting the maximum unit of service, and thereby causing fragmentation, is that otherwise it is much more difficult to *share* the resources of the intermediate node, and hence much more difficult to make the service scale up. The resources consumed by a circuit, a permanent storage allocation or an unbounded computational process can be high enough that it is not possible to provision a network to meet community needs without per-use billing. Weakening the service by imposing fragmentation means that sharing is enhanced at the expense of deterministic service guarantees.

But such weakened semantics are no more a part of the traditional model of computation than they were a part of the traditional model of storage. Conventional computational services either execute jobs to completion or perform remote procedure calls that transform inputs to outputs. Accepting fragmentation of service in LoNC means that we must view our computational service as transforming stored state within the intermediate node, perhaps completing only a part of the "job" or "call" ultimately intended by the end user. To complete the kind of unbounded computations that some applications require, multiple computational fragments will have to be applied, either using a single computational intermediate node or using many, with attendant movement of state for the purposes of fault tolerance and possibly parallelism.

An obvious objection to this kind of fragmentation of execution is that it requires active management of the computation by the end-point. It is commonly viewed as a positive aspect of execution to completion that the endpoint need not participate actively in the progress of the computation, and can in fact focus its own resources elsewhere until the result of the remote computation is required.

3.3 Statelessness

In using the best-effort services of routers to construct stronger services end-to-end, Internet endpoints rely partly on the fact that intermediate nodes are stateless, so that the construction of an alternate route affects only performance characteristics seen at the end-points. Of course storage service cannot be stateless, yet a similar philosophy was followed in the design of IBP for LoN. The only state a depot maintains is the state required in order to implement its basic services, which are allocation of storage, writing and reading. It maintains no additional state visible to the endpoint. The intent is to allow depots to appear interchangeable to the endpoint, except for the data that they actually store. Thus, although endpoints do carry the burden of keeping track of data stored at depots, they bear no additional burden of managing other visible state.

Like storage, computation (defined as transformation of stored state) can only be performed at an intermediate node that can maintain persistent state. This is the reason that computation is a natural extension of the IBP depot, and LoNC is a natural extension of LoN. However, following the philosophy of minimizing control state, the computational services implemented at the depot must not maintain any visible state other than that which they transform.

From a practical point of view, this means that there should be no state maintained in the operating system or other portions of the computational intermediate node that is required for the correct invocation of computational services. If a computational service were to change the state of the system in some way, for instance by writing data to a local temporary file, then the existence of that temporary file must not be required in order for any future computation to proceed. If file system state must be maintained between computational services, then it has to be modeled as an explicit part of the stored state transformed by the service, stored by IBP and managed by an endpoint.

Using explicit storage to maintain temporary state in a chain of computations is a burden on the endpoint, but it is not hard to understand or to incorporate into a runtime system. Another limitation imposed by statelessness is that network connections cannot be explicitly maintained between invocations of computational services. There are two approaches to dealing with this: either use a connectionless mode of communication, such as writing into and reading from a tuple space [11]; or else put the burden of communication on the endpoint, requiring it to use LoN services to implement the transfer of state between intermediate nodes. Thus, the choice is either to maintain no visible state or to make the state explicit and allow it to be managed by the endpoint.

3.4 Correctness

The use of untrusted intermediate nodes improves scalability, but it requires that the work they do be checked at the endpoint to establish correctness. In the cases of data transmission and storage, the receiver has no way to check with certainty that the data received was in fact the same as that sent; but if redundancy is added in the form of checksums, then the receiver can make a probabilistic check that the data received is *internally consistent*. This does not provide protection against malicious intermediate nodes, which can send data that is incorrect but internally consistent. It does, however, guard against errors.

One might think that settling for non-scalable network computation would make life easier because it allows computational nodes to be authenticated, and such authentication may be deemed sufficient to establish trust. But in a wide area computing infrastructure, a trusted but faulty or malicious node can cause unbounded damage. This is the result of ignoring the end-to-end principles.

The equivalent of checksums for computation is for an operation to return enough information for the result to be *verified*, at least probabilistically. A simplified example is the Greatest Common Divisor (GCD) operation: if it returns all of the prime factors of both operands, then the GCD it returns can be verified trivially. However, if only the GCD is returned, the cost of verifying it is the same as the cost of performing the operation locally. The redundancy in the results of the operation provides a check on correctness, at essentially no additional cost; and because it is deterministic rather than probabilistic, it does guard against malicious intermediate nodes.

Clearly, arbitrary operations cannot be checked in this way, and adding redundancy to complex operations may be very difficult or awkward. However, part of the research program associated with LoNC is to develop sets of operations, or "bases," for useful classes of network computation that are verifiable. These classes may be limited, but because their implementation will follow the end-to-end principles, they will not have the vulnerability to faulty computational nodes of current network computing systems based on trust.

3.5 Security

Untrusted intermediate nodes may steal or corrupt the data they handle. End-to-end security is implemented through encryption: encrypted data cannot be read or be forged without possessing the appropriate key. As in the discussion of correctness above, this works quite well in the cases of data transmission and storage because encryption commutes with both of these operations (moving and storing data). However, conventional encryption algorithms do not commute with arbitrary computations, and so in current computational systems, it is necessary to decrypt at the compute server, establishing security only between the client and that server. Again, there is no end-to-end security; the usual approach is to authenticate the server and rely on trust, with the same vulnerability.

An end-to-end approach to security in network computation would require that we use encryption that commutes with the operations implemented on the intermediate node. While it is hard to see how current strong encryption algorithms could commute with arbitrary computation, it is possible to conceive of weaker encryption (perhaps obscuring is a better word) that might commute with specific operations. A very simple example would be this: if the operation were a bitwise logical operation, like XOR, applied to large blocks of data, then permutation of the data would serve to obscure it, but the inverse permutation is simple to apply.

It is easy to see the limitations of this approach, and as with correctness, finding useful sets of operations that can be made secure in an end-to-end way is part of the LoNC research program. The thing to see is that this is the only approach that provides end-to-end guarantees for network computing. It can be combined with approaches based on authentication and trust for the sake of generality (i.e. trust but verify), however the difference in strength of the guarantees provided by the two approaches must be well understood. Any approach that is not end-to-end is vulnerable to betrayal by a trusted intermediate node.

4. THE NFU AND THE EXNODE

4.1 A Generic Network Computing Service

Before exploring the question of how to build a programmable network on the foundation of the kind of weak computational service which, as we have just seen, conformity to the end-to-end principles requires, it is important to clarify our use of the term "programmable networking." In the context of Active Networking, "programmable networking" means executing computational processes on IP routers *and maintaining process state there* in the processing of large flows, or even across multiple flows. For LoNC, on the other hand, programmable networking means *adding primitive state transformation operations* to a LoN infrastructure that already supports flexible management of persistent state, and then exposing both storage and computational resources for general use according to the end-to-end paradigm. As already noted, according to the end-to-end principles the key to achieving scalability lies in defining the right basic abstraction of the physical resource to be shared at the lowest network accessible level of the stack. For LoN the *Internet Backplane Protocol (IBP)* provides an abstraction of storage that plays this role. LoNC merely extends IBP to encompass a separate abstraction of processor resources.

To add processing power to the depots in a logistical network using this approach, LoNC must supply an abstraction of *execution layer resources* (i.e. time-sliced operating system execution services at the local level) that satisfies the twin goals of providing a generic but sharable computing service, while at the same time leaving that service as exposed as possible to serve the broadest range of purposes of application developers [5]. The execution layer, which is our term for computational service at the local level, lies below IBP in the stack shown in figure 1, at the same level as the local storage access layer. Following the familiar pattern, all stronger functions would then be built up in layers on top of this primitive abstraction.

Achieving these goals requires that the abstraction mask enough of the particularities of the execution layer resources, (e.g. fixed time slice, differing failure modes, local architecture and operating system) to enable lightweight allocations of those resources to be made by any participant in the network. The strategy for implementing this requirement is to mirror the IP paradigm. This strategy has already been used once in creating a primitive abstraction of storage in IBP to serve as a foundation for Logistical Networking [3]. That case showed how the generic service could be made independent of the particular attributes of the various underlying link or access layer services by addressing the three key features: *resource aggregation, fault detection*, and *global addressing*. Table 1 shows the results.

global communication service. IP-based aggregation of locally provisioned, link layer resources for the common purpose of universal connectivity constitutes the form of sharing that has made the Internet the foundation for a global information infrastructure. A parallel analysis applies in detail to IBP and the different underlying access layers of the network storage stack.

We call the new abstraction of computational resources at the execution layer the *Network Functional Unit (NFU)*, and implement it as an orthogonal extension to the functionality of IBP. The name "Network Functional Unit" was chosen to fit the pattern established by other components of the LoN infrastructure, which expresses an underlying vision of the network as a computing platform with exposed resources that is externally scheduled by endpoints. The archetype here is a more conventional computing network: the system bus of a single computer (historically implemented as a backplane bus), which provides a uniform fabric for storing and moving data. This was the analysis that was invoked in naming the fundamental protocol for data transfer and storage the "Internet Backplane Protocol." In extending that analogy to include computation, we looked for that component of a computer that has no part in data transfer or storage, serving only to transform data placed within its reach. The Arithmetic Logic Unit (ALU) seemed a good model, with its input and output latches serving as its only interfaces to the larger system. For this reason, we have named the component of an IBP depot that transforms data stored at that depot the *Network Functional Unit*.

Just as IP is a more abstract service based on link-layer datagram delivery, IBP's Network Functional Unit is a more abstract service based on computational fragments (e.g. OS time slices) that are managed as "operations." The independence of NFU operations from the attributes of the particular execution layer is established by working through the same features of resource aggregation, fault detection, and global addressing. Table 1 displays the results for the NFU side by side with the previous cases.

This higher level "operation" abstraction allows a uniform NFU model to be applied to computation resources globally, which is essential to creating the most important difference between execution layer computation slices and NFU operation service: *Any participant in a logistical computation network can make use of any execution layer storage resource in the network*

Table 1: Generic service abstractions for data transmission (IP), data storage (IBP), and computation (NFU).

	IP (Bandwidth)	IBP (Storage)	NFU (Computation)
Resource Aggregation	Aggregation of link layer packets masks its limits on packet size	Aggregation of access layer blocks masks the fixed block size	Aggregation of execution layer time slices masks the fixed slice size
Fault Detection	Fault detection with a single, simple failure model (faulty datagrams are dropped) masks the variety of different failure modes	Fault detection with a simple failure model (faulty byte arrays are discarded) masks the variety of different failure modes	Fault detection with a simple failure model (faulty operations terminate with unknown state for write-accessible storage) masks the variety of different failure modes
Global Addressing	Global addressing masks the differences between LAN addressing schemes and masks its reconfiguration.	Global addressing based on global IP addresses masks the difference between access layer addressing schemes.	Global depot and operation naming, based on global IP addresses and a uniform operation namespace, masks the difference between execution layer platforms

The abstractions of IP datagram service and IBP byte array service allow uniform models to be applied globally to network and storage resources respectively. In the case of IP, this means that any participant in a routed IP network can make use of any link layer connection in the network regardless of who owns it; routers aggregate individual link layer connections to create a *regardless of who owns it.* The use of IP networking to access NFU computational resources creates a global computing service.

Whatever the strengths of this application of the paradigm, however, it leads directly to two problems. First, the chronic vulnerability of IP networks and LoN to Denial of Service (DoS) attacks, on bandwidth and storage resources respectively, applies equally to the NFU's computational resources. The second problem is that the classic definition of a time slice execution service is based on execution on a local processor, so it includes strong semantics that, as we have already shown, are difficult to implement in the wide area network.

Following that line of analysis, and the model of IBP, we address both of these issues by weakening the semantics of compute resource allocation in the NFU. Most importantly, NFU allocations are by default time limited, and the time limits established by local depot policy makes the compute allocations that occur on them transient. But all the semantics of NFU operations are weaker in ways that model computation accessed over the network. Moreover, to encourage the sharing of idle resources, the NFU supports "soft" operations that use only idle cycles, as in Condor [12] and peer-to-peer systems [13]. In all cases the weak semantics mean that the level of service must be characterized statistically.

As illustrated in Figure 3, we can identify a logical progression of functionality in intermediate nodes: the router forwards datagrams, exercising control over the spatial dimension by choosing between output buffers. The depot adds control over the temporal dimension by enabling the storage of data in an IBP allocation as it passes through. Finally, the NFU is implemented as a module, added to an IBP storage depot, that transforms stored data. If we consider spatial direction, time and value to be coordinates of a single space, then the state of any data item is a point in this vector space, and the progression is one of increasing simultaneous control over multiple dimensions.[1]

Since a depot may model either disk or RAM storage resources, some NFU operations may apply only to data stored in RAM, while others may also apply to data stored on disk. Restricting an operation to data held in RAM forces any necessary movement between disk and RAM to be explicitly directed by the end-point using IBP, just as in data movement between depots. The NFU implements a single additional operation, **NFU_op**:

```
NFU_op(depot, port, operation, soft,
                          cap_1,... cap_n)
```

In this simplified form the **NFU_op** call is used to invoke an operation at the IBP depot, specified by the IP address and port it binds to. The operation is specified as an integer argument, whose meaning is set by a global registry of operation numbers. The arguments to an operation consist of a list of capabilities (cryptographically secure names) for storage allocations on the same depot where the operation is being performed. Thus, there is no implicit network communication performed by a given depot in responding to an **NFU_op** call. A flag indicates whether the operation is soft (using only idle cycles). The capabilities specified in this call can enable reading or writing, and the limitations of each are reflected in the allowed use of the underlying storage as input or output parameters. The number and type of each capability are part of the signature of the operation, specified at the time the operation number is registered. Any violation of this type information (for instance, passing a read capability for an output parameter) may cause a runtime error, but it is not checked by the implementation of **NFU_op** at the client side. Such effects as aliasing between capabilities are also not detected.

There are a number of important and obvious refinements to this call, such as handling scalar arguments and storage capabilities differently, that give it more structure and can, in some instances, make correct use and efficiency more likely. An important difference between **NFU_op** and a remote procedure call mechanism is that since the data items are assumed to be already stored in capabilities (i.e. they are already "marshaled"), all issues of data representation and type are pushed onto the operations themselves, rather than being part of the **NFU_op** call.

The behavior of the IBP depot in response to an **NFU_op** call is to map the specified capabilities into an address space, look up the operation in a library of dynamically invoked calls, and execute it. The set of calls implemented at each depot is determined by local policy. Because the storage allocations are

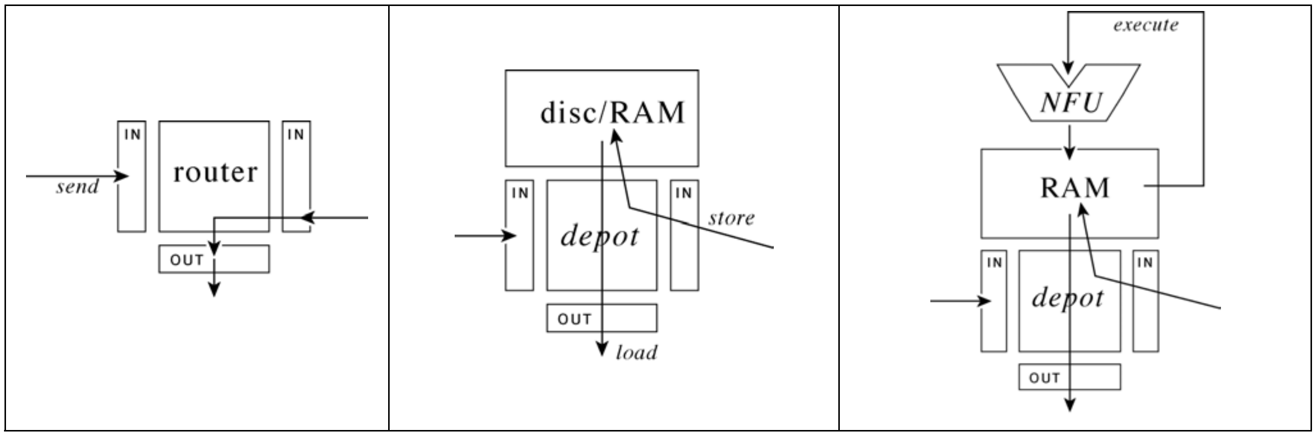

Figure 3: Intermediate nodes to manage bandwidth (IP router), storage (IBP Depot), and computation (NFU-enabled Depot)

[1] This unified point of view was nicely expressed by Dan Hillis in his 1982 paper "Why Computer Science is No Good," when he remarked that "…memory locations…are just wires turned sideways in time."[14]

local to the depot, simple memory mapping operations can be used, obviating the need for any explicit file access when invoking an operation. This limits the implementation of **`NFU_op`** to depots implemented in RAM or running on operating systems that can map files directly to a process address space. The fragmented nature of the operation can be enforced in two ways: by refusing to add to the library any call that does not provide a strict limit on resource use, and/or by monitoring resource use and terminating execution when the limit is reached. There may be operations which are of variable duration, and for which it makes sense for the user to request a long vs. a short execution, as is the case in IBP storage allocations, but we have chosen to ignore this complication here.

4.2 A Data Structure for the Flexible Aggregation of Network Computation

From the point of view of the Distributed Computing community, it is likely that one of the most striking features of the LoNC stack is the way it appears to simply jettison the well known methods of usage for computation, viz. processes invoked to run programs or execute procedure calls to completion. These familiar abstractions can be supported in the logistical paradigm, but that support must conform to its "exposed-resource" design philosophy embodied in the end-to-end principles. According to these principles, implementing abstractions with strong properties — reliability, fast access, unbounded size, unbounded duration, etc.— involves creating a construct at a higher layer that aggregates more primitive NFU operations below it, where these operations and the state they transform are often distributed at multiple locations. For example, when data is stored redundantly using a RAID-like encoding, it is fragmented and striped across depots and parity blocks are computed and stored also. All of the storage allocation and metadata representing its structure is stored in an exNode. In the event that data is lost, the exNode is consulted and data movement and XOR operations are issued to reconstruct it.

As with IBP, however, to apply the principle of aggregation to exposed computations, it is necessary *to maintain control state that represents such an aggregation*, just as sequence numbers and timers are maintained to keep track of the state of a TCP session. In the case of logistical storage allocations, we followed the traditional, well-understood model of the Unix file system's inode, the data structure used to implement aggregation disk blocks to create files, in designing a data structure called the *exNode* to manage aggregate IBP allocations [3]. Rather than aggregating blocks on a single disk volume, the exNode aggregates byte arrays in IBP depots to form something like a file. The exNode supports the creation of storage abstractions with stronger properties, such as a network file, which can be layered over IBP-based storage in a way that is completely consistent with the exposed resource approach.

The case of aggregating NFU operations to implement a complete computation is similar because, like a file, a process has a data extent the must be built up out of constituent storage resources. In a uniprocessor file system like Unix, the data structure used to implement such aggregation is the process control block, which includes both a memory map component for RAM resources and a swap map component for disk resources. In fact, process extents and files are very closely related, as can be seen by the existence of system calls like `mmap` that identify the storage extent of a file with part of the data extent of a process [10].

Exposing the primitive nature of RAM and disk as storage resources has the simplifying effect of unifying the data extent of a file with the data extent of a process; both can be described by the exNode (Fig. 4). But each must be augmented with additional state to implement either a full-fledged network file or a full-fledged network process. Thus, the closely related services of file caching, backup, and replication, on the one hand, and process paging, checkpoint, and migration, on the other, can be unified in a single set of state management tools.

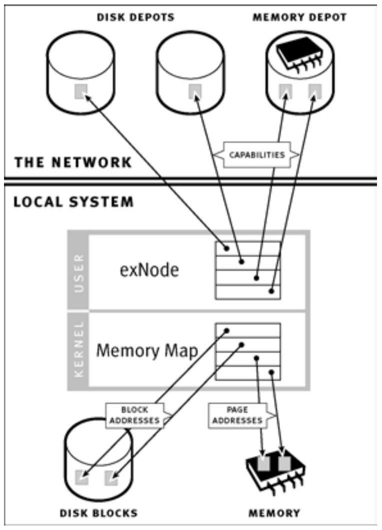

Figure 4: The exNode provides a uniform view of data and process state in LoNC

4.3 Pipelining for NFU performance

At this point, the reader may wonder how we propose to obtain any degree of performance out of such a fragmented service, a service where each time slice must be individually allocated, and where data movement and control must reside at a possibly distant network end-system. In truth, this is one more component of our research program, but one that we believe we have powerful tools to address. In this section, we briefly sketch the analogy we see between the execution of NFU operations in the wide area and the execution of a stored program on a pipelined RISC processor architecture.

Any computation or service can be characterized as taking an initial state S_0 and applying a series of primitive transformations t_0, t_1,… t_n, generating a sequence of intermediate states $S_i = t_i(S_{i-1})$ and a final state S_n. When the computation takes place on a network intermediate node, such as an NFU-enabled depot, there is an issue of how that sequence of primitive transformations is to be organized and effectively invoked.

One approach is to associate a local generator for the entire sequence of operations with a "service" and invoke that service using a single name $t_i = G(S_i)$. A benefit of this strategy is that the sequence of transformations can be generated with no latency and applied directly to the state stored on the intermediate node. Its weakness is that the intermediate state S_i is generally bound to the intermediate node, due to non-portable encoding and storage of data and transformations. Also, by implementing the generator at the intermediate node the set of available services is defined by what the operator of that node is willing to load.

Another approach is to invoke the sequence of transformations explicitly, making each primitive operation an individual "service." In this model, the generator must be used remotely to determine the next transformation to apply. The advantage of this strategy is that it is easier to define the primitive transformations in a portable way, at least for certain classes of transformations. By moving the generator to the edge, the particular services that can be built out of those transformations are not dependent on the operator of the intermediate node, within certain bounds on the utilization of node resources. The disadvantage of this approach comes from the latency in the application of the generator and the issuing of primitive transformations, as well as from the overhead of issuing a network operation for every primitive operation.

The form of the problem being confronted here is the same as that of a well-known problem in the history of computer architecture: how to generate and issue a stream of micro-operations to drive a RISC processor. In that case, two important features were used to address issues of latency and overhead for streams of fine-grained operations: pipelining and caching.

- A standard approach to overcoming the latency of issuing instructions is to pipeline them: later instructions are sent before responses have been received from earlier instructions.
- When the generation function G is highly regular, it is possible to store a portion of it locally on the intermediate node and then to substitute local generation for remote generation in some cases. With caching, this is done automatically by introducing two modifications to the scheme:
    - The entire generation function is modeled as a stored program and program counter.
    - Control is reversed, with the remote generator being invoked explicitly by the intermediate node (the FETCH operation).

Thus, by using pipelining, and perhaps even a stored program model with caching, we believe that the performance gained through uninterrupted operation of a stand-alone process executing at the intermediate node can be granted to the NFU when it conforms to the requirements of the end-system. But the end system would never relinquish the discretion to limit such independence and exert application-specific control.

4.4 NFU Scenarios

Creating a generic network computing service that conforms to end-to-end principles would be a mere curiosity unless there were real applications that could make use of it. Although we feel that various compelling applications for the NFU can be found in areas such as multimedia content distribution, digital libraries, distributed database design, and Grid services for scientific computing, we here discuss only a few relatively simple examples.

4.4.1 Filtering a stream of frames

An active service that can be easily implemented using the NFU is a filter that applies a test to every frame in a stream of frames and discards those that fail. This example is stateless, involving only the remote application of the test through a call to the NFU and action on the result.

But such a simple operation can be progressively elaborated. A more complex version eliminates duplicate frames by always storing the previous frame in a persistent buffer. The per-frame test compares the value of the stored frame to the current frame. Whenever the current frame differs from the previous frame, it becomes the new stored frame and the old value of the stored fame is discarded.

A still more complex version stores the initial frame and compresses the stream by sending only a difference function computed between the stored frame and the current frame. The stored frame is used until a difference is encountered that is larger than some fixed threshold. At that point, the entire current frame is sent and it becomes the new stored frame.

In all cases, a controlling program at a network endpoint is responsible for moving the stream of frames through the depot and invoking the NFU operations that apply the test and, in the last case, compute the difference function. A persistent controlling state of the network process exists at the edge, represented by an exNode and controlled by a conventional process.

A measure of fault tolerance can be achieved in this scenario by writing the stored frame redundantly to multiple depots whenever it changes. Furthermore, by maintaining a copy of frames sent to the filtering depot at the sender until they have been filtered and forwarded, failure at the depot can be recovered through recovery of state and restarting any NFU operations whose result state has been lost. The type of redundant state management and the resulting degree of fault tolerance are the responsibility of, and are under the control of, the endpoint program.

4.4.2 Merging streams of records

Consider several streams of records being produced by multiple senders and merged, in order, at intermediate nodes according to some record comparison operation. In this case, a conventional approach would be to locate a merge process at each intermediate node and have the nodes communicate to implement the flow control necessary to keep their queues from overflowing. Sessions between intermediate nodes that are so established have state that is stored at the nodes and is not externally visible.

One of the key problems in conventional treatments of this kind of application is that the end user has to establish the authority to start the necessary merge process at each operation. Given that the process encapsulates not only the notion of comparison required to implement the merge, but also the policies for flow control, it is likely to be application-specific rather than generic. This means that it must be expressed as a program that executes at the intermediate node. But if the problem of establishing the trust necessary for such execution can be overcome, the merge can be implemented.

However, an issue immediately arises concerning reliability: if any intermediate node fails, the data and session state that were located at that node, or in transit to or from it, at the time of failure will be lost. What is required is a more complex scheme for managing intermediate state. Enough redundancy must be maintained in the stored state of the merge tree to recover from a node failure, and the control state required to make use of that stored state must be accessible outside of any one node. This issue can be addressed in the design of a merge process that is implemented at each intermediate node, but it becomes a difficult problem in distributed control. LoNC, however, offers a clear alternative.

In order to implement the merge using LoNC, it is only necessary to have the basic comparison and merge operations available through the NFU. Because these are such generic operations, there is a good chance that they can be implemented in forms that would be reusable across broad classes of applications and so could be installed semi-statically. The strategy of exposing general (if not completely generic) operations relies on the assumption that this would allow substantial sharing of operations without the need for application-specific trusted code.

An endpoint that implemented the merge using NFU operations for comparison and merging would manage the entire state of the merge tree as it flowed from sources to sink. Every buffer full of data would reside in an IBP allocation, as would any intermediate state that might be maintained and communicated between merge operations. Use of disk as well as RAM resources to buffer larger amounts of data when required to adjust for differences in flow, and the algorithms used to prefetch data back to RAM in anticipation of its being used in an operation, would be under the control of the endpoint, as would all replication of data and control state required to support fault tolerance.

4.4.3 Edge services

The simplest model of Web and streaming media is to deliver the same service to clients anywhere in the network from a centralized server or a set of replicas. A slightly more complex model has a centralized server deliver a generic service to an edge device, or "middlebox," that then reprocesses it to create the ultimate end-user service. Examples include the transcoding of video streams between formats or bitrates, and the construction of personalized Web pages using databases of user-specific information. One problem with the deployability of this architecture is that it requires that middleboxes be ubiquitous within the network and that the necessary business arrangements be made between the operators of servers and the operators of middlebox infrastructures.

When the creation of a standard middlebox framework was addressed by the IETF OPES Application Area Working Group, the IAB was very concerned about the interposition of an active agent between the client and server [15, 16]. The need to maintain control and accountability gave rise to many caveats regarding the requirement that one of the two endpoints authorize any edge service, and the need to alert the client when an edge service is applied to the output of the server. The fundamental problem is that the introduction into the system of an agent that is neither the client nor the server creates the possibility that both would act in the interest of some third party whose services have not been explicitly invoked.

Given that edge services are simple enough to be implemented using atomic operations, LoNC makes it possible for either the server, or the client, or both, to implement edge services using the NFU. The question of authorization becomes moot, since there is no active agent other than the client and the server.

4.4.4 Distributed data queries

The Web has given rise to a highly decentralized mode of data management in which many individual producers store their data locally in a manner that is globally visible. Then, indexing services such as search engines scan the visible data and create indices against which later queries are executed. The creation of global indices is a massive job, requiring supercomputer-scale processing power and storage capacity. For this reason, querying of global data is only possible through expensive and unwieldy infrastructures such as those deployed by Google and other search engines, and queries that fall outside of the scope of their indices are impossible for the average user or project. Other examples of such global query infrastructures are peer-to-peer [17, 18].

An alternative architecture is to allow ad hoc queries to be made directly against distributed data, using a massively parallel NFU infrastructure that is located on or close to the systems that store the data. In this model, indices can be constructed on the fly and maintained for as long as they are useful to a single user or to a community. The level of resources required to build global indices is hard to manage only when it is centralized and owned by a single project or company. The scale required to manage substantial subcollections of the entire Web will be the size that fits easily into a globally provisioned LoNC infrastructure.

5. CONCLUSION

In this paper we have shown how Logistical Networking can be extended beyond its origin as an end-to-end approach to network storage to include network computation. The result combines data transfer (bandwidth), data persistence (storage) and data transformation (computation) in a uniform model of state management that, because it is designed for scalability using the end-to-end paradigm, can be applied to many difficult problems in wide area computation. Yet the very fact that this approach tries to synthesize all these elements in one common scheme has led some to question whether or not this work belongs to the field of Networking at all, rather than some other field of distributed systems. We have two answers to this query, which, taken together, help to put these ideas in a wider perspective.

In the first place, Logistical Networking borrows from the fundamental principles of Wide Area Networking, which seems to us to be the reason that it has always been best understood by the networking community. One way to characterize the field of Wide Area Networking is to say that it encompasses the subset of distributed systems that is restricted to those that scale globally. In other areas of Distributed Systems, such as distributed databases or telecollaboration, scalability is a *virtue* that designers strive for, but a system that is less scalable can still hope to fill an important niche. By contrast, in Wide Area Networking scalability has been taken as a *fundamental requirement* for architectural acceptability from the outset of the design process.

Logistical Networking likewise embraces scalability as a defining characteristic. We seek to broaden the interpretation of the end-to-end principles, which we view as necessary guides to achieving scalability, to services other than the undelayed and unmodified delivery of bits. By putting those principles first, we have derived a design for ubiquitous infrastructure that has the familiar limitations of weak semantics, but which includes storage and computation and, arguably, will scale.

In the second place, Logistical Networking seeks to create a unified framework by exploiting the commonality that exists between storage, networking and computation; it therefore draws on possibilities that fall outside any one these fields when they are considered in isolation from one another. At the heart of the underlying analysis is an unremarkable observation: *the state of every computer system consists of data that is stored, except when it is either propagating along a datapath or being transformed by a processor.*

From this common starting point, however, the engineering of systems has diverged:

- Storage focuses on buffers that can maintain large amounts of data for long periods of time.
- Networking focuses on the "fast path" that passes data quickly between wide area data paths.

- Computing focuses on very powerful mechanisms for transforming stored state.

Based on these differences in implementation, practitioners in each area have come to distinguish their fields in terms of constructs that do not exist in the others:

- Storage systems implement "files" that can be stored indefinitely with great fidelity.
- Networks implement "datagrams" that have a source, a destination and a speed of transmission.
- Computing implements "processes" that have an initial state and inputs, maintain intermediate state, and produce output.

Constructs such as these have been highly valuable as an aid in dealing with the complexity of the underlying hardware systems and the huge numbers of possible states that they can assume. At times, implementations that enshrine these constructs have been seen as the key to achieving performance, by allowing the optimization of interactions between components along the critical path.

Yet sometimes such abstractions constrain the thinking and the implementation strategies of architects. History shows that when this happens, the way to new functionality, or even to the highest performance, can lie in looking past the constructs to the components, and working directly with them. Examples of such successful reductionism abound: load/store architecture (e.g. CDC 6600), primitive OS kernels (e.g. Unix), RISC architecture (e.g. MIPS) and store-and-forward wide area networking (e.g. IP). In each case, functionality is moved from a lower layer to a higher layer of the infrastructure, with a greater burden of scheduling and control being placed on the latter. Once the opportunity for architectural innovation is identified, the question is not whether it is "right" or "wrong," but whether it has significant advantages from a systems engineering point of view.

Logistical Network Computing views all computer systems as data stored in buffers that can be maintained, or moved to other (possibly distant) buffers, or transformed. From this point of view, a block of data sitting on a disk can be treated as a long-lived datagram that is not currently going anywhere; and similarly a datagram passing through a router can be seen as a process whose pages extend over space but are not transformed. By understanding the commonality between these constructs, we hope to define a unified framework without artificial boundaries or "balkanization." If this approach were to succeed, would such a unified field of endeavor, focused on scalability as its principle design criterion, be what we now call "Networking?" Our answer is yes, but it would also be much more.

6. Related Work

Logistical Network Computing touches on most aspects of resource management for wide area distributed systems, and so it has an overlap with most Distributed Operating System projects. The important categories are remote job execution [19], remote procedure call [20], state replication for fault tolerance and mobile code and agent infrastructures [21]. While this list is not exhaustive, space restrictions prevent us from giving a more complete account.

The area of Active Networks [4] is obviously very closely related to Logistical Networking because it also seeks to use the storage and computational resources of intermediate nodes to implement innovative services. As we have discussed at various points in the paper, the difference between the two is that Active Networks takes the step of placing an unbounded process execution at an intermediate node, which has the effect, predictable by reference to the end-to-end principles, of limiting the scalability of the system. Avoiding this compromise is the defining goal of Logistical Networking.

Calvert, Griffioen and Wen [22] have developed *Ephemeral State Processing* as a mechanism to maintain persistent state at IP routers and perform operations on it. As with Logistical Network Computing, they followed the design principles of IP to create an architecture that conforms to the end-to-end principles: storage allocations are limited in size and duration, instructions are restricted to a limited set installed on the router, and both functions are best effort. However the scale of their ephemeral state is orders of magnitude smaller than the storage supported by Logistical Network Computing: storage allocations are limited to single 64 bit words stored for 10 seconds; primitive operations analogous to individual machine instructions act on one or two stored words. While this greatly reduces the problem of scalability, it also restricts the applicability of their approach to very simple services.

The other area of Distributed Systems that comes closest to the principles and methods of Logistical Networking is peer-to-peer. Peer-to-peer computing [13] systems differ from Logistical Network Computing because they tend to be application-specific and therefore not to be appropriate for deployment on common infrastructure. Each application — SETI@home, folding@home, etc.— distributes its own computational processes to run on end user workstations, and so creates a separate non-interoperable infrastructure. Attempts to create generic peer-to-peer computing platforms, such as Entropia's, run in to the problem that the mobile code that runs on it must be trusted, so scalability is limited to corporate intranets.

7. Acknowledgements

The authors would like to acknowledge the work of Alex Bassi, Jeremy Millar and Yong Zheng in the prototyping and implementation of the NFU-enabled depot.

8. REFERENCES

[1] D. Atkins, K. Droegemeier, S. Feldman, H. Garcia-Molina, M. Klein, P. Messina, D. Messerschmitt, J. Ostriker, and M. Wright, "Revolutionizing Science and Engineering through Cyberinfrastructure: Report of the National Science Foundation Blue-Ribbon Panel on Cyberinfrastructure," Panel Report, January, 2003. http://www.communitytechnology.org/nsf_ci_report/.

[2] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-End Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277-288, November, 1984.

[3] M. Beck, T. Moore, and J. S. Plank, "An End-to-end Approach to Globally Scalable Network Storage," in *Proceedings of ACM Sigcomm 2002*. Pittsburgh, PA: Association for Computing Machinery, 2002.

[4] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wethrall, and G. J. Minden, "A Survey of Active Network Research," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80-86, January 1997, 1997.

[5] D. P. Reed, J. H. Saltzer, and D. D. Clark, "Comment on Active Networking and End-to-End Arguments,"

IEEE Network, vol. 12, no. 3, pp. 69-71, May/June, 1998.

[6] M. Beck, T. Moore, J. Plank, and M. Swany, "Logistical Networking: Sharing More Than the Wires," in *Active Middleware Services*, vol. 583, *The Kluwer International Series in Engineering and Computer Science*, S. Hariri, C. Lee, and C. Raghavendra, Eds. Boston: Kluwer Academic Publishers, 2000.

[7] J. S. Plank, A. Bassi, M. Beck, T. Moore, M. Swany, and R. Wolski, "Managing Data Storage in the Network," *IEEE Internet Computing*, vol. 5, no. 5, pp. 50-58, September/October, 2001. http://computer.org/internet/ic2001/w5toc.htm.

[8] J. Carter, P. Cao, M. Dahlin, M. Scott, M. Shapiro, and W. Zwaenepoel, "Distributed State Management," Report of the NSF Workshop on Future Directions for Systems Research,, July 31, 1997. http://www.cs.utah.edu/~retrac/nsf-workshop-report.html.

[9] J. Dennis and E. V. Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143-155, March, 1966.

[10] A. Bassi, M. Beck, and T. Moore, "Mobile Management of Network Files," in *Third Annual International Workshop on Active Middleware Services (AMS 2001)*. San Franscisco, CA: Kluwer Academic Publishers, 2001, pp. 106-115.

[11] N. Carriero and D. Gelernter, "Linda in Context," *Communications of the ACM*, vol. 32, no. 4, pp. 444-459,1989.

[12] D. H. J. Epema, M. Livny, R. v. Dantzig, X. Evers, and J. Pruyne, "A worldwide flock of condors : Load sharing among workstation clusters," *Journal on Future Generations of Computer Systems*, vol. 12 1996.

[13] A. Oram, "Peer-to-Peer: Harnessing the Power of Disruptive Technologies." Sebastopol, CA: O'Reilly & Associates, 2001, pp. 448.

[14] W. D. Hillis, "New Computer Architectures and Their Relationship to Physics or Why Computer Science is No Good," *International Journal of Theoretical Physics*, vol. 21, no. 3/4, pp. 255-262,1982.

[15] A. Barbir and A. Rousskov, "OPES Treatment of IAB Considerations," IETF, Internet Draft, draft-ietf-opes-iab-00, June 12, 2003. http://www.ietf.org/internet-drafts/draft-ietf-opes-iab-00.txt.

[16] D. D. Clark, J. Wroclawski, K. Sollins, and R. Braden, "Tussle in Cyberspace: Defining Tomorrow's Internet," in *Proceedings of ACM Sigcomm 2002*. Pittsburgh, PA: Association for Computing Machinery, 2002.

[17] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz, "Maintenance-Free Global Data Storage," *IEEE Internet Computing*, vol. 5, no. 5, pp. 40-49, September/October, 2001. http://computer.org/internet/ic2001/w5toc.htm.

[18] D. Anderson and J. Kubiatowicz, "The Worldwide Computer," *Scientific American*, vol. 286, no. 3, pp. 40-47, March, 2002.

[19] I. Foster and C. Kesselman, "The Grid: Blueprint for a New Computing Infrastructure," Morgan Kaufman Publishers, 1999, pp. 677.

[20] R. Srinivasan, "Remote Proceedure Call Protocol Specification, Version 2," IETF, RFC 1831, August, 1995. http://www.ietf.org/rfc/rfc1831.txt?number=1831.

[21] E. A. Brewer, R. H. Katz, E. Amir, H. Balakrishnan, Y. Chawathe, A. Fox, S. D. Gribble, T. Hodes, G. Nguyen, V. Padmanabhan, M. Stemm, S. Seshan, and T. Henderson, "A Network Architecture for Heterogeneous Mobile Computing," *IEEE Personal Communications Magazine*, vol. 5, no. 5, pp. 8-24, October, 1998.

[22] K. L. Calvert, J. Griffioen, and S. Wen, "Lightweight Network Support for Scalable End-to-End Services," in *Proceedings of ACM Sigcomm 2002*. Pittsburgh, PA: Association for Computing Machinery, 2002.