Micah Beck, Huadong Liu, Terry Moore and Yong Zheng

Pipelining and Caching the Internet Backplane Protocol

Logistical Computing and Internetworking Laboratory
Computer Science Department, University of Tennessee
Knoxville, TN 37996-3450 USA

Abstract-- In this paper, we show how pipelining and caching can be used to optimize the performance of the Internet Backplane Protocol, an overlay implementation of the Transnetworking architecture that offers exposed data transfer, storage and processing resources. Our experimental results clearly indicate that this approach can yield high performance in operations that require high utilization of intermediate node resources.

1. INTRODUCTION

The Internet Backplane Protocol (IBP) is the essential mechanism of Logistical Networking (LN), which is an overlay implementation of a new architectural approach to highly generic, interoperable computer networking. The fundamental idea behind LN is that the end-to-end principle that has guided the development of the Internet also applies to other types of shared information infrastructure, specifically to infrastructure that includes substantial storage and processing resources. Because it incorporates such resources, which in conventional network applications are provisioned at hosts or endpoints, the scope of LN is significantly broader than that of traditional network protocols. It encompasses application areas previously addressed by programmable networking approaches like Active Networking [10, 17, 26], Ephemeral State Processing [8] or even many overlay, multimedia, storage or content distribution networking schemes [1, 12, 16, 21]. LN seeks to provide a shared infrastructure that supports functionality which overlaps areas of wide area distributed systems that are typically thought to lie outside the domain of networking, such as distributed visualization and data mining, database management, and even Grid computing.

The central tenet of Logistical Networking is that the end-to-end principle, which has served so well as a guide to the design of shared infrastructure for the transfer of data between endpoints (which is how we characterize IP datagram delivery service), can be applied, with analogous advantages, to the design of shared infrastructure for storage and processing services. The motivation for generalizing the idea of shared infrastructure in this way is the same as one expressed by the authors of the original end-to-end arguments: *application autonomy* [20]. What LN questions, however, is the view that applications can best achieve the desired freedom to change as needed by sharing *only* data transfer services and implementing all other resource management at endpoints. The Internet Backplane Protocol is our tool for modeling services based on the management of generalized shared resources.

But despite the fact that it shares many of the goals of and has achieved a reasonable degree of early acceptance by the networking community [5, 6], LN faces three significant objections that impede further engagement by a broader community. First, some have objected that a network comprised of LN's "fat" intermediate nodes, provisioned with substantial storage and processing resources, would run foul of the end-to-end principle, and so could not possibly scale. Our interpretation of the end-to-end principle (detailed below), which casts the design principles of the Internet as special cases of those of more general systems, provides an explanation of why IBP can scale. We believe that the current deployment of IBP as a global storage service bears out this

This work is supported by the National Science Foundation (NSF) Next Generation Software Program under grant # ACI-0204007 and NSF Middleware Program under ANI-0222945, the Department of Energy Scientific Discovery through Advanced Computing Program under grant # DE-FC02-01ER25465, and with additional support from the Center for Information Technology Research (CITR) of the University of Tennessee. The infrastructure used in this work was supported by the NSF CISE Research Infrastructure program, EIA-9972889 and Research Resources program EIA-022444.

analysis. Second, it is sometimes claimed that the steps that must be taken to consider the end-to-end principle (mainly the weakening of the semantics of the basic LN storage and processing services and the use of untrusted intermediate nodes to provide them) would render the resulting service unusable to real applications. However, the diligent construction of end-to-end protocols and tools has made it possible to develop services with semantics strong enough to meet, and in some cases exceed, the requirements for performance and reliability of the application communities that have adopted LN, including content distribution, remote visualization, video transcoding, and data mining.

Finally, it has been objected that LN's steadfast adherence to mechanisms that take account of the end-to-end principle, requiring that detailed control over the functioning of intermediate nodes be managed by endpoints, cannot possibly provide adequate performance in applications where the sequence of operations to be performed is dynamically determined during the execution of an application protocol. The reason is obvious: in the wide area network, latencies between endpoints and intermediate nodes can be long enough that delays due to synchronous signaling cannot be tolerated. In this paper we address this objection by showing how the techniques used to obtain performance from processors in architectures with high instruction issue latency — pipelining and caching of operations at the point of execution — can be applied to the IBP operations executing at the intermediate node.

While Logistical Networking was developed, and has been presented here, as a generalization of conventional Internetworking aimed at serving application requirements for distributed storage and processing, we have also come to understand it, in a more fundamental way, as a model of how to generalize the ability of the intermediate node to make use of link layer network technologies. In order to provide context for our experimental results, in Section 2, we summarize this new architectural analysis. It embodies an approach to achieving greater interoperability in networking by exposing the capabilities of the intermediate node through a common interface at a new layer of the network stack, called the *transit layer*, which we propose to insert between the current link and network layers. We describe the IBP as an overlay implementation of transit layer functionality and represent it as prototype of a generalized type of interoperable sharing of information infrastructure we have termed *Transnetworking*. In Section 3 we present the IBP and its implementation on intermediate nodes called *depots*. In Section 4 we introduce mechanisms for pipelining and caching IBP and work through an example in detail of implementing a high performance merge at a remote depot using these mechanisms. In Section 5 we discuss issues of fault tolerance and correctness, and in Section 6 we present related work.

2. LOGISTICAL NETWORKING: TRANSNETWORKING ARCHITECTURE IN OVERLAY FORM

The end-to-end principle was developed as a methodology for understanding the impact of assigning functionality to the layers of a communication stack in the delivery of data across a shared infrastructure [23]. Although its precise meaning is still a matter of debate [18], the small cluster of notions it encompasses is widely acknowledged to have had a pervasive influence over the development of the Internet's architecture [9]. We call the interpretation of the end-to-end principle that has guided the evolution of LN, and which we believe captures the key idea expressed in the Internet's design, the *scalability interpretation*. According to this view, the end-to-end principle represents a hypothesis about fundamental limits on the scalability of services that are designed to be deployed on a shared network infrastructure. By "scalability" we mean deployment scalability, i.e. a property representing the relationship between an increase in the "size" of a network or distributed system's deployment (as measured, for example, by the number of devices attached, intermediate nodes deployed, networks subsumed, organizational borders crossed, applications supported, etc.), and the concomitant changes in the size or degree of other attributes considered important to the system's success, such as performance, reliability, and operating cost. Given these elements, the end-to-end principle can be given a general formulation as follows:

The end-to-end principle: When designing a service that is implemented using a shared infrastructure, there is an inherent tradeoff between the service's scalability on that infrastructure, and both i) its specialization to a particular class of applications, and ii) the value or scarcity of the resource consumed to provide it.

Now in a scalable network, such as the Internet, the shared infrastructure is composed of links and intermediate nodes. The basic services of the network are built from the resources provided by those elements. In our view, there are three primitive services that network intermediate nodes can make available as resources to enable the creation of network services: transferring data between neighboring nodes, storing data, and applying transformational operations to data. The end-to-end principle, as we interpret it, embodies the idea that there is a complex structural constraint on the relationship between the way in which these services are implemented on a distributed system and the system's deployment scalability, and that this constraint expresses itself in a set of tradeoffs that network designers inevitably have to make. Since a primary design goal of LN was maximum deployment scalability, this view largely determined the design of IBP's primitive storage and computation services, as we have described in detail elsewhere [5, 6].

But although the model of IP has been critical to informing the design of IBP, and the existence of the IP network has been indispensable to implementing LN services that can be widely deployed for serious testing, the relative success of that effort suggests that this child of the general Internet design philosophy might offer a path to an even higher level of scalable interoperability than the Internet currently provides. For if the end-to-end principle can be successfully applied to storage and processor resources "in the network," as we believe IBP shows that it can be, then that argues for the possibility of a common service that is more general than simple datagram delivery, and therefore able to address a broader range of application requirements than IP can.

To see how this might be so, consider the primitive services that network intermediate nodes could make available for the creation of other network services: transfer of data between neighboring nodes, storage of data locally, and processing of data locally. Arguably these three basic services are at an architectural level that is below the network layer of the current stack, since they are required in order to implement end-to-end IP datagram delivery. IP service is specialized in the sense it exposes only data transfer, but encapsulates the other two fundamental services in order to implement optimized packet forwarding. However, if we look at the basic services that applications want, but which IP doesn't model, storage and processing of data in transit, i.e. at intermediate nodes, are among the most important.

Our early work on LN focused on the area of overlay content distribution networks for large data objects, and it developed by way of a succession of software design and implementation cycles, with user feedback from a variety of application communities. As a result of this previous work, we are able begin our exploration of the performance potential of the Transnet architecture with a relatively full and robust version of transit layer functionality in the form of the LN software suite, implemented in overlay on TCP/IP and Linux.

Unlike IP datagram delivery, storage and processing resources are inherently local to the intermediate node, rather than being defined across a homogeneous network layer. In that respect, they are more analogous to the link layer, which connects adjacent nodes. Thus, as an initial step, we propose to generalize the view of layer 2 to include local storage and processing services. We call this more general layer, which includes link, storage and processing as coequal elements, the local layer in Figure 1.

The local layer exhibits the same extreme heterogeneity as the link layer in each of its three elements or dimensions. A single intermediate node may support storage resources as different as fast access RAM and large disk arrays; processing services can be implemented on commodity microprocessors or FPGAs. The key point to notice, however, is that all of the operations of the local layer can be modeled as providing services of various kinds to arrays of bytes of data that are stored, transiently, at the intermediate node.

For this reason, a network protocol capable of expressing a broad class of operations on byte arrays located at the intermediate node can abstract a more general class of lower layer services than IP does. By choosing to model operations that are either local to the intermediate node or restricted to operating on nodes connected by adjacent links, the protocol can avoid implementing routing or wide area algorithms of any kind. Because network layer protocols, can in fact be implemented on top of it, we would situate the new protocol between the

local and network layers in the current network stack. Since a protocol at this layer would provide an abstraction of services for data that is “in transit” at the intermediate node, we propose to call it the transit layer in Figure 1.

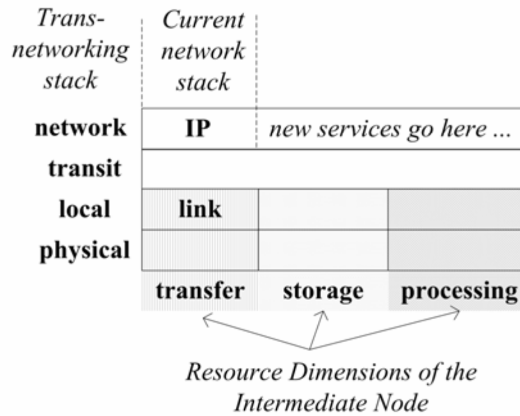


Figure 1: The Transnet stack compared to the traditional network stack

Of course it is one thing to argue, in a somewhat a priori way, for transit networking as a new architectural paradigm, but something quite different to show that that paradigm can actually deliver on its promises, especially with regard to the difficult challenge of creating a scalable network services that can also deliver the kind of performance that some advanced application demand. Fortunately the ideas for Transnetworking and a transit layer protocol arose directly out of our work in LN and IBP, which essentially constitute an implementation of transit networking functionality overlaid on the current IP model. Since LN already has a substantial software base and a globally deployed testbed, it provides an ideal vehicle for exploring, as we do below, the performance issues that can be expected for services designed to conform to the stringent requirements of the end-to-end principle. It is worth noting, however, that if Transnetworking, in this overlay implementation, can be shown to be capable of delivering reasonable performance results while doing valuable work, then it should be clear that the potential impact of doing a low-level, transit layer implementation and deployment of IBP is likely to be immense.

Our view is that a transit layer protocol that abstracts the particular characteristics of different technologies at the local layer, while being more general and sitting below the network layer in the stack, would exhibit greater deployment scalability and provide a broader foundation for network interoperability than IP. We call the architectural approach based on such a protocol, and on the interpretation of the end-to-end paradigm that should inform its progressive development, Transnetworking.

Transnetworking stack	*Logistical networking’s overlay implementation*
transport	**LoRS**
network	**L-bone**
transit	**IBP (overlay)**
local	**TCP & Linux**
physical	**physical**

Figure 2: Comparison of the Transnet architecture and it’s overlay implementation

The key enabling element is the Internet Backplane Protocol (IBP), which is a highly generic, best effort, network storage service. It was designed to model the full resources of a network intermediate node in isolation from the network paths it might connect, and was engineered according to the end-to-end principle in order to maximize scalability. As illustrated in Figure 2, IBP implements the transit layer functionality of a common storage service, and is the foundational layer of the logistical networking stack [2, 19]. As described below, the processing dimension of the transit layer has been added to IBP through a modular extension to the protocol called the Network Functional Unit (NFU), which provides an abstraction of processor resources of the intermediate node [5].

3. IBP AND THE NETWORK FUNCTIONAL UNIT

In order to take account of the end-to-end principle, both the design of IBP and the NFU closely follow the model of IP datagram delivery and are designed as generic, best effort services. IBP provides a more abstract storage service based on blocks of data (on disk, memory or other media) that are managed as "byte arrays." To allow a uniform model to be applied to storage resources generally, IBP masks the details of the local disk storage: instead of fixed block sizes, it aggregates blocks at the local layer; instead of a variety of failure modes, faulty byte arrays are simply discarded; instead of diverse addressing schemes, it provides a large, uniform, randomized capability name space which serves to insulate users from one another [11]. Likewise, the NFU provides more abstract compute service based on computational fragments (e.g. OS time slices) that are managed as "operations" applied to data stored in IBP memory buffers. The NFU abstracts the details of the processing platform at the local layer: instead of offering fixed OS time slices, it aggregates them; instead of a multiplicity of failure modes, faulty operations terminate with unknown state for write-accessible storage; instead of a multiplicity of processing resources, the NFU offers a set of "op codes" in a uniform operation name space. Thus, both IBP and the NFU provide a transit layer abstraction of local layer resources of LN's overlay intermediate node, or depot.

But although such generic abstractions help secure the kind of interoperability that the transit layer is intended to provide, achieving scalability requires that the semantics of the service also be as weak as possible, i.e. both IBP and the NFU must offer best effort services. In IBP, this means that 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, and such "admission decisions" can be based on both size and duration. Similarly, NFU allocations are time limited, and the time limits established by local depot policy makes the compute allocations that occur on them transient. Thus, at the transit layer, storage and computation are essentially "packetized," and the multiplexing of the service must be characterized statistically.

To maintain scalability, all stronger services— availability, reliability, fast access, unbounded size, correctness, security, etc. — must be implemented on top the transit layer from the end points. A key to enabling such services is the possibility of aggregating either IBP or NFU allocations, and this in turn requires a mechanism to hold the metadata and control the state of such aggregations. In LN, this role is played by the exNode [3, 6], which is modeled on the familiar Unix inode, but applied to IBP storage resources. For IBP, the exNode aggregates byte arrays in IBP depots to form something like a file. 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. 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. However, 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/replication and process paging/checkpoint/ migration can be unified into a single set of state management tools.

IBP depots are servers on which clients perform operations on remote storage via RPC. As Transnetworking's common buffer service, the fundamental categories of operations implemented by IBP are: 1) Allocate a storage

buffer on an intermediate node; 2) transfer data between buffers; 3) process data stored in some set of buffers under the control of a specific predefined operation. Storage allocation has added complexity here because of the creation of persistent state visible to the client and the need to manage it in a manner that protects clients from one another (using capabilities) and that allows the intermediate node not to overtax its storage resources (best effort leases, maximum allocation size). Data transfer and processing are completely stateless, manifesting their effects solely by modifying the values stored in their storage buffer arguments.

The IBP depot can leverage a variety of data transport mechanisms between nodes, and this is a key part of the common buffer service that provides the interoperability that Transnetworking is designed to achieve. Both data transfer between IBP depots and NFU calls are parameterized by an operation code which specifies one of a number of alternative modules to implement the operations. Data transfer modules, called Data Movers [4], implement different protocols for moving data between depots, including standard TCP, UDP multicast, and SABUL, a specialized high throughput protocol based on UDP [14]. NFU modules implement fragments of computation that operate on data stored in IBP allocations.

The LN middleware that has so far been built to use the transit layer functionality of IBP depots focuses on storage and implements limited network layer functionality at network endpoints in overlay style. The Logistical Runtime System (LoRS) consists of a set of tools and associated APIs that, by allowing users to draw on a pool of depots, enable the implementation of files and other storage abstractions with a wide range of characteristics, such as large size (through fragmentation), fast access (through caching), and reliability (through replication). Operations, such as replication and deletion, can be combined to provide more complex functionality, such as simple routing. LoRS tools also implement some transport layer services such as checksums, encryption, and compression and erasure codes, all of which is implemented at the end-points. Along with command line and GUI interfaces, LoRS provides an extensive, low-level C API that provides developers with even greater flexibility.

4. Pipelining and Caching IBP

The design intent of IBP, as described above, is to maximize the generality of the class of services that can be provided by intermediate nodes, or depot, by exposing a model of the services of those nodes that is very fine-grained, being expressed in terms of individual buffer transfers. In order to take account of the end-to-end principle, control over the execution of these fine-grained operations must be retained by the endpoint, and delegated to the depot only through an explicit operation that allows control to revert to the endpoint as needed.

The initial protocol design of IBP was indeed intentionally naïve, with each operation implemented by a Remote Procedure Call transmitted using a per-call TCP connection (RPC-over-TCP). The intent of adopting such a simple design approach was both to preserve simplicity of early implementation and to avoid pre-optimization before gaining experience. Early applications of IBP storage and movement of large amounts of data are inherently parallel, and so it was possible to achieve high levels of performance through the use of multithreading in the client.

While this approach imposed an overhead due in the creation of a TCP connections and the management of threads, it worked well enough to serve the needs of substantial user communities. However, performance issues did arise in applications requiring the forwarding of data at the depot [4], and this early design was never able to achieve high performance in the presence of such flow data dependences, which had to be resolved through synchronization at the client. The addition of computation to IBP, implemented in the form of NFU operations performed on data in IBP storage allocations, introduces further dependences between operations that impose substantial sequentiality.

In this paper, we describe two extensions to the IBP protocol that are designed to overcome the cost of fine-grained control of the depot by the client: operation pipelining and instruction caching. The new version of the

protocol, IBP version 2.0, that supports both of these features, is not currently distributed except for strictly experimental purposes.

4.1. Pipelining IBP Operations

A straightforward step in the evolutionary optimization of IBP was to allow *multiple operations* to be issued from client to depot using a single persistent TCP connection, thus amortizing the cost of TCP connection setup. Persistent connections are available in IBP version 1.4, which is currently available in a beta release. Second, on a single TCP connection, IBP now implements *pipelining*: later operations can be issued before earlier operations have completed, up to a pipeline depth that is settable by agreement between client and depot.

These two extensions required a redesign of the lower layers of the IBP protocol, to create an asynchronous interface that enables operations to be issued and responses to be received separately. The protocol can support out-of-order responses from the depot in order to maximize performance and minimize the use of buffer resources at the depot, using per-operation tags to keep operations correctly associated with their responses. Dependences between operations are detected dynamically and enforced by the depot.

The introduction of persistent connections and pipelining between client and depot has the effect of creating state at the depot that is leveraged by the client over the execution of multiple operations. This state includes the session state of the TCP connection as well as the synchronization state of sequential operations in the pipeline. The total depot resources consumed by a connection increases with the duration of the connection and with the depth of the pipeline. For this reason, our end-to-end design approach requires that the depot be allowed to place limits on the duration of connections and the depth of pipelining. As with all aspects of IBP service, a persistent connection to an IBP depot must be considered best effort service, and subject to failure, and any operations whose response has not been received at the time of a connection failure will have unknown result.

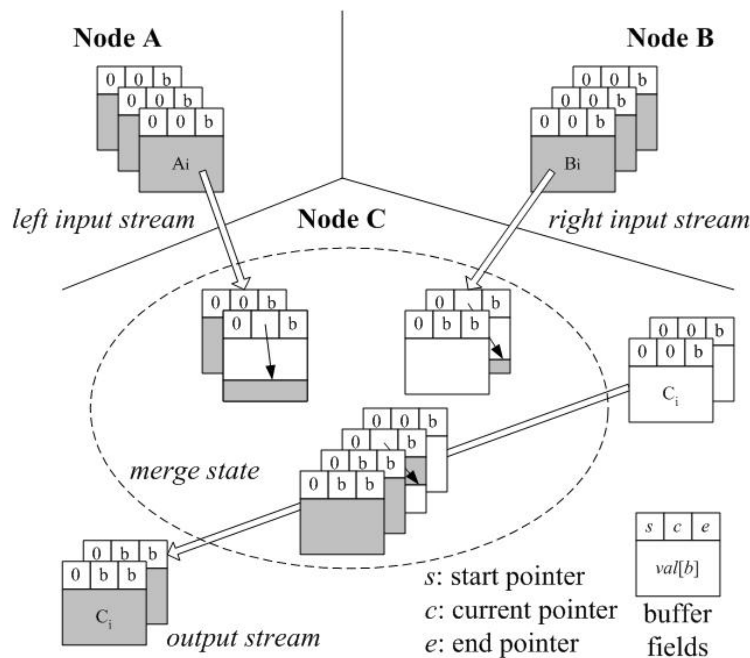


Figure 3. Global flow of buffers in the NFU merge example

In order to demonstrate the power of pipelining to increase the performance of network applications implemented using the NFU, we now present, as an extended example, a description of how this approach can be used to perform a merge operation on ordered data streams distributed in the wide area network. Consider, then, a scenario in which two large streams of linearly ordered data records originate from different locations, A and B,

within the wide area network, and a single ordered collection, consisting of the merged contents of both input streams, has to be generated and stored at a third location, C. As shown in Figure 3, the merging of the streams is a multistage process that requires the movement of data across the wide area from both A and B to C, as well as the application of a record-level comparison and data movement operations at C in order to generate the result.

In order to implement this process using depot operations, we must first characterize it in terms of more primitive operations on fixed-size buffers of data that can be moved by making `IBP_copy` calls and processed by making `NFU_op` calls. For our purposes, each buffer consists of a sequence of data records, plus an additional control record which is used to hold the state of the merge process itself. This decision to combine the representation of data and control state into a single buffer simplifies our example, but requires that the data stream representation be adapted to fit this particular algorithm.

Each merge block M is a fixed-size vector of b records, *M.val[b]*, plus three pointer values, *M.start*, *M.current* and *M.end*. These pointers are initialized to 0, 0 and b respectively. A simple non-pipelined `merge` operation operates on three merge blocks M_l, M_r and M_o, the first two representing the current blocks in the two streams (left and right) being merged, and the last representing the partially generated next block in the merged result stream. $M_l.current$ is compared with $M_l.start$ and $M_l.end$. If $M_l.current >= M_l.start$ and $M_l.current < M_l.end$, then M_l has data available to be used in this merge operation. The same check is applied on M_r, and and analogous check ensures that space is available in M_o for the result. At the end of any `merge` call, one of the input buffers M_l or M_r is exhausted, or the output buffer M_o is full.

Using this simple `merge` operation, the merge process can be expressed as a sequence of `merge` call, each of which exhausts an input buffer or fills an output buffer, or both. In response to each call, the client issues a new call, replacing an empty buffer with the next full buffer in the input stream or replacing a full output buffer with an empty one and issuing the next `merge` call. The movement of data from nodes A and B to C is also directed by the client in response to the results of each `merge` call, with an appropriate level of read-ahead implemented to keep data flowing.

The obvious problem with this simple scheme is that the client must intervene between `merge` calls in order to issue data transfer and `merge` operations. Even though the client may have initiated read-ahead sufficient to place the necessary input buffers at node C for the next operation to proceed, the operation cannot be initiated except by the client. Thus, this highly data-dependent sequence of operations has a collateral sequence of *control* dependences that are being resolved at the client, imposing latencies between operations of one or more round-trip times between the client and node C.

The complication of additional control dependence can be overcome by a combination of techniques used in processor pipelining: *loop unrolling* and *speculative issuing of instructions*. The first step is to expand inputs and outputs of the `merge` operation, allowing multiple blocks in each input stream and in the output stream to be represented in a single operation. The most basic form of expansion represents two input blocks from each stream, but we will in fact represent four blocks in the output stream in order to balance inputs with outputs. The purpose of this expansion is not to allow multiple blocks to be processed by a single operation; each operation still terminates as soon as an input block is exhausted or an output block is filled. Instead, its purpose is to enable enough ambiguity in the location of data accessed by the `merge` operations to transform control dependence into data dependence.

When a pipelined merge instruction `NFU_op(merge, CA0, CA1, CB0, CB1, C0, C1, C2, C3)` arrives at the depot, *CA0.current* is compared with *CA0.start* and *CA0.end*. If $CA0.current \geq CA0.start$ and $CA0.current < CA0.end$, *CA0* is used as the left merge input in this merge operation. If not, pointers of *CA1* are similarly checked, and if data is available it is used. Otherwise, the merge operation is interpreted as a no-op. The same checking is performed on the streams from B and C to select the right merge input and the merge output respectively, and the pointers of the merge blocks operated on are updated by the `merge` operation.

Because each stream is represented by multiple input blocks, while multiple output blocks are also available, successive operations can proceed *without the intervention of the client*. With the addition of more input and output blocks, it becomes possible for one `merge` operation to compute the pointer values to be used in the next `merge` operation at the same time as its result is being returned to the client. Thus, as long as neither input stream exhausts all of its available blocks and the output buffers are not full, *the same merge operation can be restarted up to five times without client intervention*. Once an input stream has exhausted all of its available input blocks or all the output blocks are full, no further `merge` work can be done.

Using the `merge` operation in this form to implement a pipeline of depth *d*, the client issues *d* identical `merge` operations, on the assumption that the data will be evenly enough distributed to allow several operations to be executed without exhausting all of the input blocks available on either stream. Each time a block is emptied or filled, the result returned to the client specifies which has occurred, and the next operation issued reflects the result of the previous operation. Anytime that the pipeline drains due to an unbalanced flow of data between the two input streams, speculative operations are issued by the client to refill it.

To see how this works, let us refer to the IBP capabilities for the inputs streams of blocks stored at A and B and the output stream of blocks generated and stored at C by labels Ai, Bi and Ci respectively. Then an example execution might proceed as follows (the contents of the operation pipeline at representative steps are shown in Table 1):

Table 1. Operations in the pipeline at representative steps

Step	**Operations in Pipeline**
2	`NFU_op(merge, CA0, CA1, CB0, CB1, C0, C1, C2, C3)`
	`NFU_op(merge, CA0, CA1, CB0, CB1, C0, C1, C2, C3)`
	`NFU_op(merge, CA0, CA1, CB0, CB1, C0, C1, C2, C3)`
	`NFU_op(merge, CA0, CA1, CB0, CB1, C0, C1, C2, C3)`
4	`NFU_op(merge, CA0, CA1, CB0, CB1, C0, C1, C2, C3)`
	`NFU_op(merge, CA0, CA1, CB0, CB1, C0, C1, C2, C3)`
	`NFU_op(merge, CA0, CA1, CB0, CB1, C0, C1, C2, C3)`
7	`NFU_op(merge, CA0, CA1, CB0, CB1, C1, C2, C3, C4)`
	`NFU_op(merge, CA0, CA1, CB0, CB1, C0, C1, C2, C3)`
	`NFU_op(merge, CA0, CA1, CB0, CB1, C0, C1, C2, C3)`
10	`NFU_op(merge, CA1, CA2, CB0, CB1, C1, C2, C3, C4)`
	`NFU_op(merge, CA0, CA1, CB0, CB1, C1, C2, C3, C4)`
	`NFU_op(merge, CA0, CA1, CB0, CB1, C0, C1, C2, C3)`
11	`NFU_op(merge, CA1, CA2, CB1, CB2, C2, C3, C4, C5)`
	`NFU_op(merge, CA1, CA2, CB0, CB1, C1, C2, C3, C4)`
	`NFU_op(merge, CA0, CA1, CB0, CB1, C1, C2, C3, C4)`
	`NFU_op(merge, CA0, CA1, CB0, CB1, C0, C1, C2, C3)`

Legend Grayed cells: exhausted or full buffers;

Inverted cells: active buffers;

1. Choosing a data read-ahead of depth 8, the client issues `IBP_copy` operations to move A0-A7, and B0-B7 to temporary buffers at C, which we will refer to as CA0-7 and CB0-7, respectively.
2. Choosing an instruction pipeline depth of 4, the client waits until the contents of CA0-1 and CB0-1 are available, and then issues four identical NFU `merge` operations: `NFU_op(merge, CA0, CA1, CB0, CB1, C0, C1, C2, C3)`.

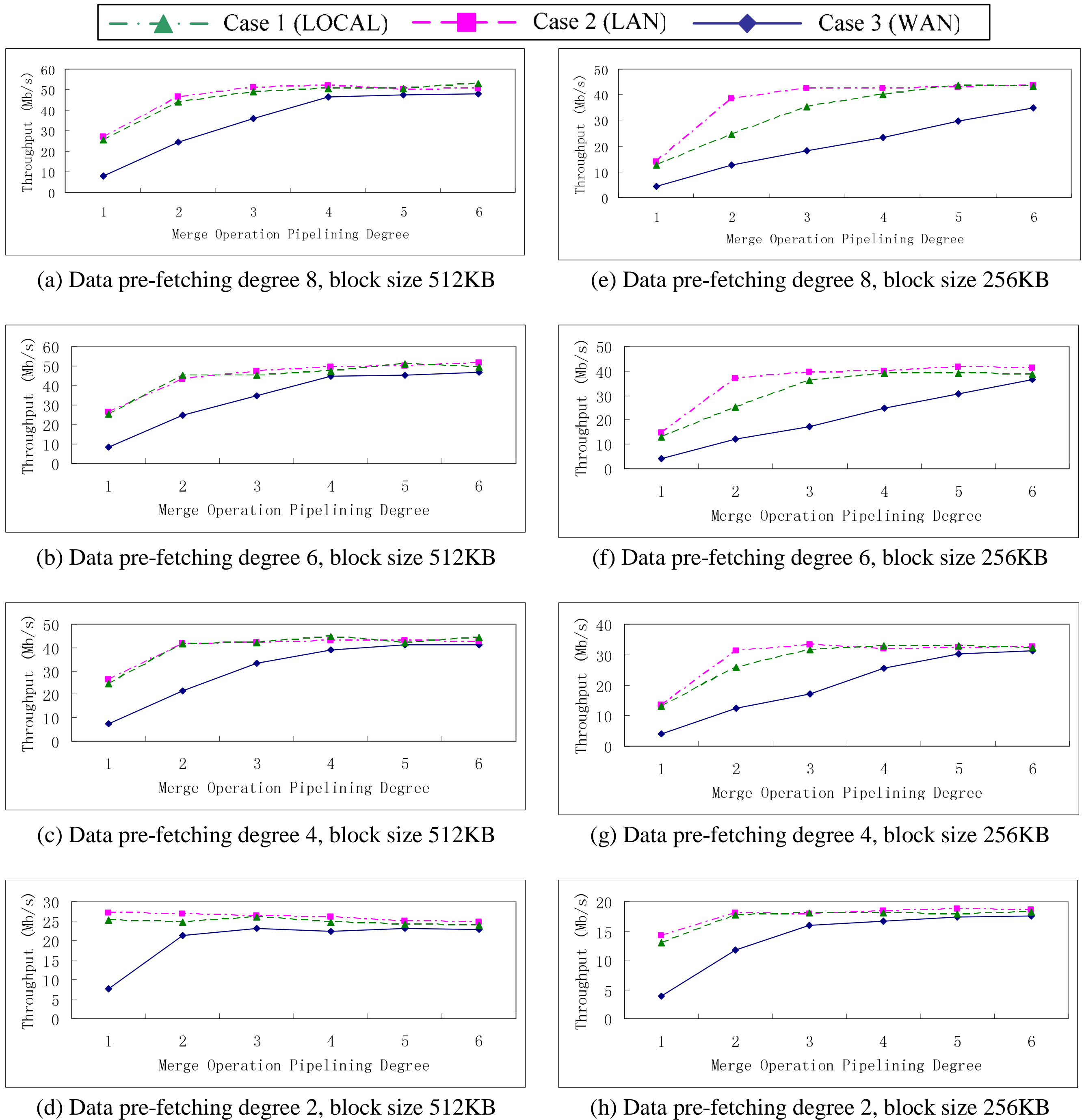
3. If the output block C0 becomes full, the first `merge` operation will terminate and a result indicating this fact will be sent to the client.
4. The second `merge` operation will proceed, generating merged data to C1.
5. When the client receives the result of the first `merge` operation, it will issue a fifth `merge` operation: `NFU_op (merge, CA0, CA1, CB0, CB1, C1, C2, C3, C4).`
6. If the temporary input buffer CA0 is exhausted next, the second `merge` operation will terminate and the result indicating this status will be sent to the client.
7. The third `merge` operation will proceed, drawing data from temporary input buffer CA1.
8. When the client receives the result of the second `merge` operation, it will wait until the contents of CA2 are available and then issue a sixth `merge` operation: `NFU_op (merge, CA1, CA2, CB0, CB1, C1, C2, C3, C4)` as well as issuing an `IBP_copy` operation to transfer A9 to CA9.
9. If temporary input buffer CB0 is exhausted and output block C1 becomes full, the third `merge` operation will terminate and result indicating this fact will be sent to the client.
10. The fourth `merge` operation will proceed, drawing data from CA1 and CB1, and generating data to C2.
11. When the client receives the result of the third `merge` operation, it will wait until the contents of CB2 are available and then issue a seventh `merge` operation: `NFU_op(merge, CA1, CA2, CB1, CB2, C2, C3, C4, C5)` as well as issuing an `IBP_copy` operation to transfer B9 to CB9.

A set of experiments were run using an IBP depot located at the Starlight network collocation facility in Chicago and the Abilene NOC on the Indiana University/Purdue University campus in Indianapolis as nodes A and B, and a depot located on the Knoxville campus of the University of Tennessee as node C. The degree of read-ahead/pre-fetching on the input streams (pipelining between the client and nodes A and B) varied from 2 to 8, and the degree of pipelining of `merge` operations (between the client and node C) varied between 1 and 6. Experiments were then run with three different client locations, from

- Case 1: Being co-resident with the depot that implements the `merge` operation on node C (thus approximating the degree of coupling between issue and execution that is present in encapsulated solutions such as active routers).
- Case 2: Running on a host in the same local area network as node C.
- Case 3: Running on a host at the San Diego Super-computing Center.

The record used in all experiments was a single four-byte integer. Experiments were performed with 512KB and 256KB input/output blocks respectively (thus the size of a block in records *b* were 64K and 128K). The experimental results of 512KB blocks are plotted in Figure 4.a to 4.d and those of 256KB blocks are plotted in Figure 4.e to 4.h. Each experiment was run 9 times, with the result reported being the average. We summarize these results as follows:

- As would be expected, the performance of the merge operation increased with increasing block size and degree of read-ahead. Parallelism in the implementation of read-ahead may have led to the exploitation of more simultaneous TCP streams as the degree of read-ahead increased.
- In all of the scenarios, the performance of the sequential cases (merge operation pipeline depth 1) was lower when the client was separated from node C by the wide area network (case 3) than in the other two cases, generally by a factor of 3-4.
- In all cases, a pipeline depth of 6 was sufficient to yield merge performance with the client separated from node C by the wide area network (case 3) equivalent to the other two cases, and in some cases this was achieved with a pipeline depth of 4.
- One unexpected result was that in some cases, the performance was slightly better when the client was located on another machine in same LAN as node C (case 2) than when the client was co-resident on node C (case 1). We believe that this is due to paging or other contention for resources between the depot and the client when they are on the same node.



(a) Data pre-fetching degree 8, block size 512KB

(e) Data pre-fetching degree 8, block size 256KB

(b) Data pre-fetching degree 6, block size 512KB

(f) Data pre-fetching degree 6, block size 256KB

(c) Data pre-fetching degree 4, block size 512KB

(g) Data pre-fetching degree 4, block size 256KB

(d) Data pre-fetching degree 2, block size 512KB

(h) Data pre-fetching degree 2, block size 256KB

Figure 4: Merge throughput as a function of operation pipeline degree

4.2. Caching IBP Instructions

The experiments described in the previous subsection make use of instruction pipelining to overcome latencies caused by the separation of client and depot in the wide area. However, our experience with pipelining has shown us two things:

1. The complexity and specificity of NFU operations required to support pipelining are increased when techniques such as loop unrolling described above are used to hide deep pipelining. This approach breaks down when the dynamic resolution of control dependences is not predictable.
2. When implementing highly repetitive processes such as stream processing, pipelining requires that IBP operations that are identical or that differ only in the identity of buffers being processed be continually resent from the client.

For these reasons it is desirable in the case of our merge example, and necessary when faced with more complex dynamic behavior, to delegate more autonomy to the depot so that control dependences can be resolved at the depot. We have done this by extending the semantics of the IBP protocol with a stored instruction model and implementing an instruction cache at the depot.

We define a labeled IBP *instruction* is to be a pair *<op, successors>* consisting of a NFU operation, *op*, and a specifier, *successors,* of a set of possible successor instructions within the instruction cache. When allowed to issue instructions autonomously, the depot can issue another instruction from within the set specified by *successors*, the choice being determined by a return value generated as the result of executing *op*. A new IBP operation `IBP_istore (`*icap, instr*`)` is defined which associates an IBP instruction *instr* with a particular label *l* in an instruction cache specified by a capability *icap*.

Introducing greater autonomy into the issue of IBP operations requires that proper consideration be given to application of the end-to-end principle in this context. It would be a simple matter to allow the client to download a program to the depot cache, and then initiate the execution of a process in the style of an active networking router or other dynamically extensible encapsulated network service [25]. However, such a strategy would generate encapsulated process state at the depot that is inaccessible to the end system. Instead, we choose to adopt an approach which allows the client to grant autonomy to the end system in a limited way that is subject to continual monitoring by the endpoint and requires renewal in order to continue. The degree of autonomy granted to the depot at any time is under the complete control of the client, and can be modified dynamically according to the nature of the computation and the changing conditions in the network and execution environments.

Given that the `IBP_istore` operation has been used to load the instruction cache, each connection between an IBP client and depot is considered to have a session state consisting of a number of *instruction issue credits.* The client can increase/decrease the number of issue credits by an integer *n* by issuing a special operation `IBP_credit(`*n*`)`. The meaning of the issue credits is that if the IBP client issues a special operation `IBP_start(`*l*`)`, then the instruction with the specified label *l* will be executed, and the issue credits will be decreased by one. The successor instruction will then be issued, and autonomous issue will continue until all the issue credits are consumed. As each instruction completes, its result is returned to the client, just as if the instruction had been issued by the client itself. At that point where all issue credits have been depleted, issue stops. This allows a task consisting of a fixed number of IBP instructions *n* to be initiated by setting the issue credits to *n* number and then issuing an `IBP_start` operation.

While the issue credits are nonzero, the client may grant further autonomy to the depot by means of additional `IBP_credit` operations, which can be performed during the execution of such a task, prolonging the execution and deepening its pipeline indefinitely. At the extreme, a special call sets the issue credit value to infinity granting the depot full autonomy to execute processes to completion (until an instruction is executed which has no successors). In order to lessen the burden of supervision on the client, further options will be added in future to allow the client to suppress return values or otherwise control the communication between depot and client. Any such options reducing the direct role of the client in execution at the depot would be exercised at the discretion of the client, with the option always being available to return to a highly supervised or even synchronous mode of operation. Thus, autonomy is granted by the client to the depot in a graduated way, when the client feels that it is necessary and advisable in view of factors such as congested or faulty behavior of the wide area network.

This mechanism allows the client to obtain performance when it is available without sacrificing control when it is needed. Note that issue credits granted by the client allow but do not obligate the depot to continue issuing instructions from the cache. The depot may, in order to preserve its own resources, choose not to issue instructions at the earliest opportunity, and can even reject operations issued directly by the client, allow the pipeline to drain and close the client connection. However, accepting a given pipeline depth is considered an agreement by the depot to make a best-effort attempt to complete operations that have already been issued.

Having introduced a stored instruction model at the depot provides the client with a mechanism to delegate the resolution of control dependences between IBP operations to the depot, and overcomes the problem of the client needing to resend a stream of identical operations when implementing a highly repetitive process. However, some highly repetitive processes implemented at IBP depots, particularly those that operate on data stored on disk, do not operate on a fixed set of IBP allocations, but rather on a stream of allocations. In order to implement such processes with a fixed set of instructions, it is necessary to represent a dynamic set of operations operating on a stream of allocations, which we have accomplished by storing arrays of IBP capabilities in other IBP capabilities and using index variables and array referencing to choose the specific capabilities to operate on in each iteration. Each array of capabilities has two integer indices associated with it, one to indicate the length of the array, and the other representing offset of the next active capability for processing.

In addition to processing arrays of capabilities representing stored data, the need arises to handle multiple capabilities in a uniform manner when data transfers implemented as `IBP_copy` operations between depots are pipelined. In this case, it is necessary to be able to initiate asynchronous send operations and to later block on receiving their responses. This is analogous to the notion of a split phase transactions in dataflow architectures [15]. The additional state required to implement this synchronization is implemented as a transaction control block that has one entry for each outstanding data transfer. The control block is stored in an IBP capability that is read and written by the depot as part of managing this form of the `IBP_copy` operation. The designs of both the capability array and transaction buffer mechanism described here are preliminary, and have been developed to handle the specific examples described here. We expect to revise these designs to provide more generality, particularly in use of indirection and the manipulation of index variables, as we gain experience in using them.

The cached version of our merge example is shown in Figure 5. Note that the `INC` operation is used to increase the index of the next active capability for processing in the capability array. We can see that the core of our merge is comprised of a small set of instructions which executes in a highly repetitive fashion, using only a single NFU operation and attendant data movement calls. In an environment where data transfer is reliable, it is easy to imagine the execution of this process being very efficient, and techniques such as spot compilation or other architectural support being used to allow it to execute at very high speed, given a sufficient budget of credits. However, in an environment where faults are also possible, or where network conditions might change, the client might choose to periodically save the state of the merge operation to some other node. Also, if data transmission errors were to occur, this might result in an error being reported back to the client, requiring corrective action that is not part of the core merge loop. Both of these cases require the client to be able to take control of the merge process.

We ran a sequence of experiments in which we simulated the effect of pipelined execution of our merge when the client was separated from node C by the wide area network (case 3) using the IBP instruction cache and issue credits. After issuing some initial data movement operations to prime the data pipeline, the client would issue a number of credits allowing the depot to issue that number of instructions from the cache. Then, as the program executes, credits are issued to allow the continued execution, maintaining approximately the same number of outstanding credits. Note that small variations in the number of credits issued are made to allow for differences in the length of conditional branches (see Figure 5).

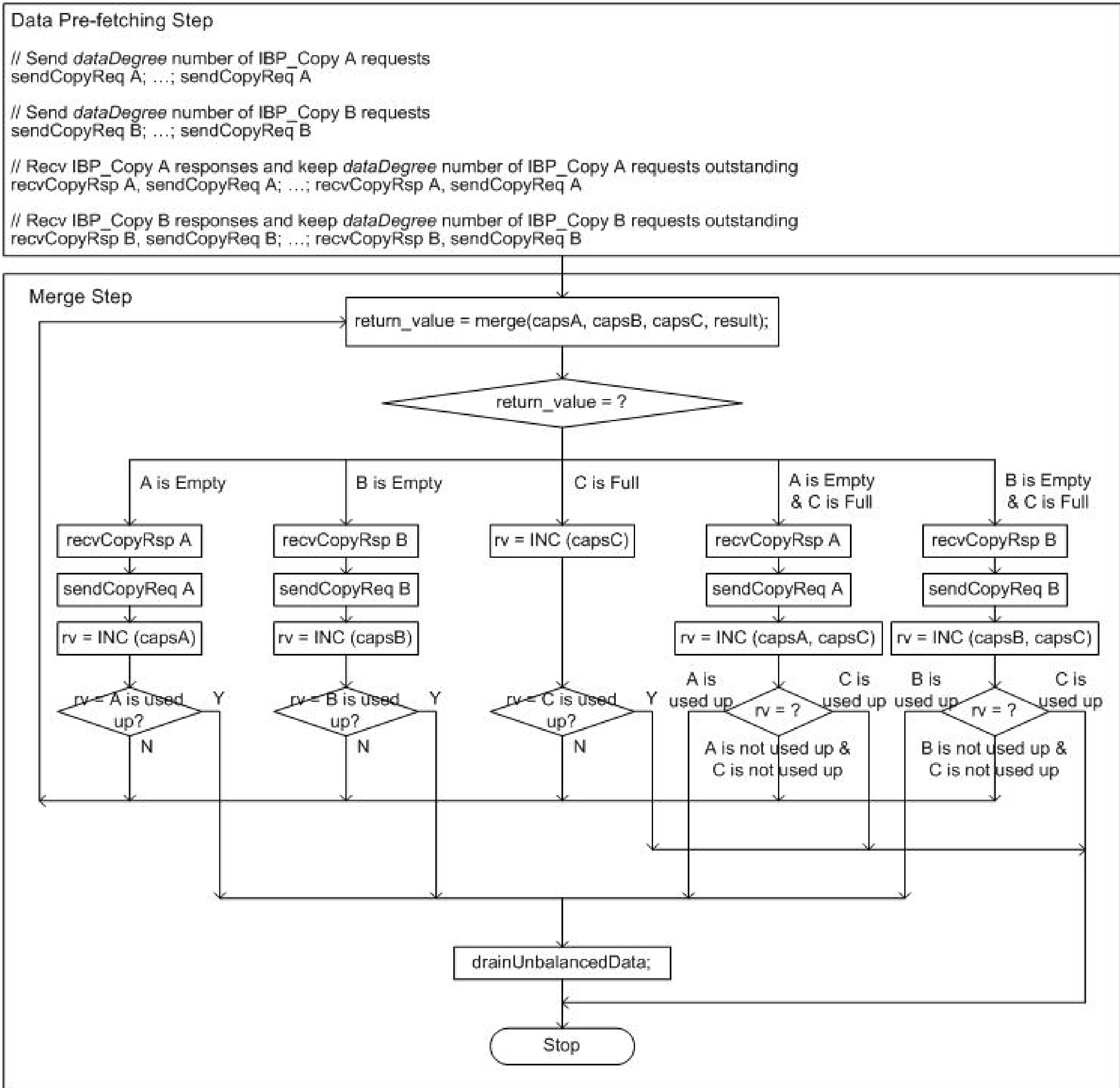
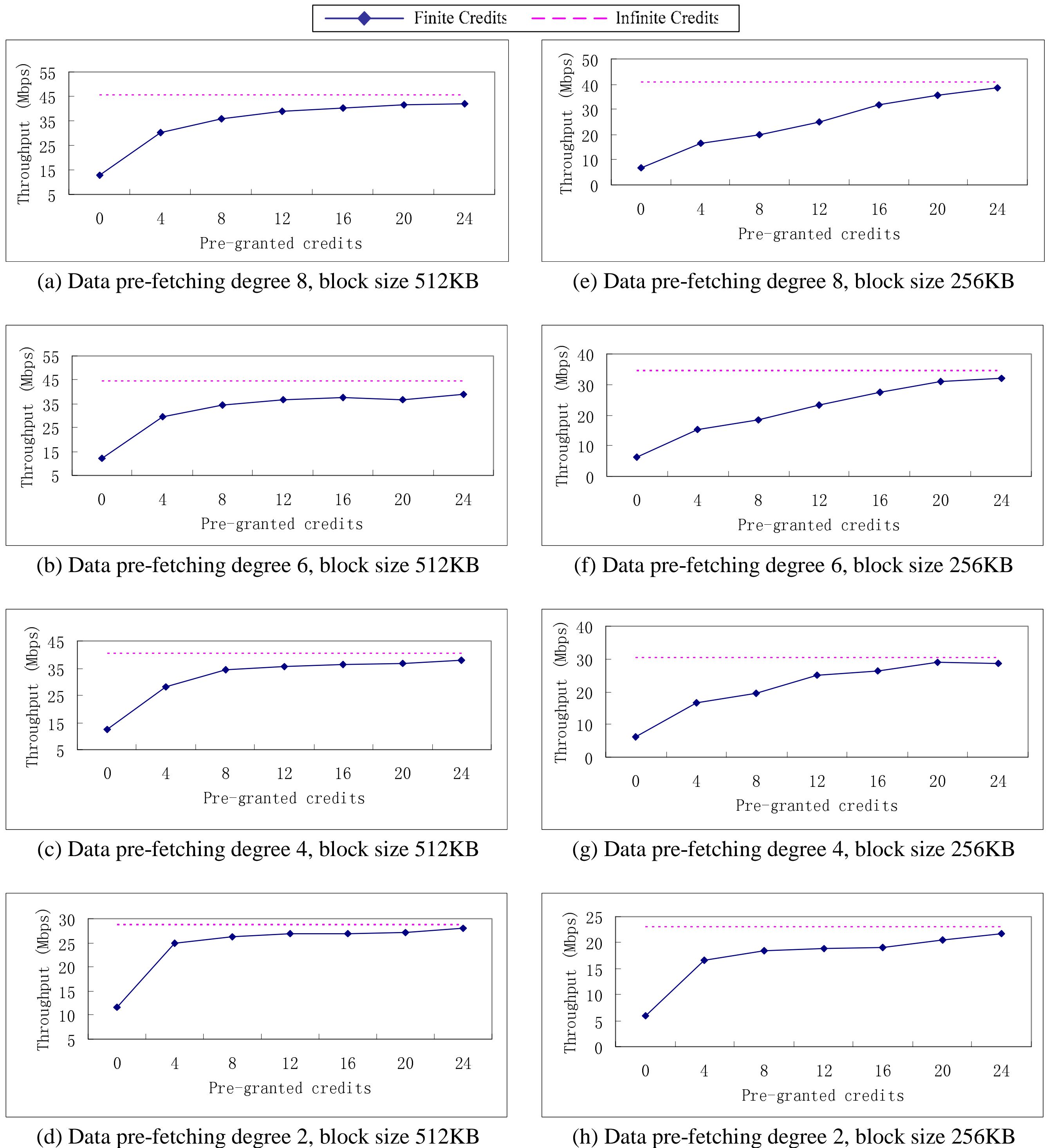


Figure 5: Flowchart of the cached merge example

Figure 6.a to 6.d and Figure 6.e to 6.h show the merge throughputs of our cached merge example running with 512KB and 256KB input/output blocks respectively. Each data point represents an averaging of the results of 9 runs. The line marked with "infinite credits" represents the performance when more than the number of credits required for the entire merge process to complete is issued initially. For example, in the case of block size 512KB with data pre-fetching degree 8, 1920 credits are issued initially. Because several instructions must be issued corresponding to each pipelined operation, due to the use of asynchronous data movement, credits and pipeline depth are proportional but not equal. In that case, there is no synchronization between the client and depot.

Careful readers may have noticed that almost in all cases, performance of the pipelined merge with degree 6 is slightly better than the cached merge with infinite credits. This is because all execution status of cached

instructions (sending copy request, receiving copy response, increasing capability index, and merging, see Figure 5) are returned to the remote client. When responses except that of the `merge` instruction were suppressed, throughput of cached merge with infinite credits outperformed that of pipelined merge as expected.



(a) Data pre-fetching degree 8, block size 512KB

(e) Data pre-fetching degree 8, block size 256KB

(b) Data pre-fetching degree 6, block size 512KB

(f) Data pre-fetching degree 6, block size 256KB

(c) Data pre-fetching degree 4, block size 512KB

(g) Data pre-fetching degree 4, block size 256KB

(d) Data pre-fetching degree 2, block size 512KB

(h) Data pre-fetching degree 2, block size 256KB

Figure 6: Merge throughput vs. excess credits issued

5. OTHER ISSUES

This paper seeks to explore the practical aspects of implementing high performance protocols and services using the Transnetworking architectural model. From that point of view, our experiments have focused on overcoming the obstacles that taking an end-to-end approach to network storage and computation places in obtaining high performance. We have ignored many of the advantages and the great flexibility that the Transnetworking architecture offers, because those are the subject of other papers and investigations.

To be fair, it is not just the benefits of the end-to-end approach that we have ignored, but also problems and issues other than those of performance due to latency in signaling between client and depot. High on the list of issues are security and correctness, which are easily addressed in networking and storage applications by using end-to-end protocols. The use of untrusted intermediate nodes to implement network processing is another matter altogether.

In this section we focus on two important issues, in an effort to convey to the reader who is unfamiliar with Logistical Networking the flexibility that the architecture offers as well as the challenges still to be faced.

5.1. Fault Tolerance

The difficulty of overcoming faults is one of the great challenges of networking and distributed programming of all kinds. The end-to-end principle requires that the transit layer be not only generic but also *best effort*, which means that any operation can fail, and that there can be few assumptions regarding the state of any part of the network from which a message is not received confirming that state. However, in our examples, we have not taken account of what happens when a communication or computational operation fails.

We get away with this omission because the implementation of IBP is an overlay, and uses a TCP connection as a reliable umbilical between client and depot. Our examples cannot be considered complete, and our deployment will not scale even using TCP connections, until we have built up sufficient services to overcome realistically occurring errors. (In a somewhat unfortunate dissonance of terminology, the deployment of protocols to overcome local-layer failures at IBP depots implements “hop-by-hop” rather than “end-to-end” reliability, although it is under the control of the endpoint (the client) and the implementation should be scalable, since it takes account of the end-to-end principle.) The implementation of end-to-end distributed storage services based on IBP in the Logistical Runtime System is quite mature, but experimentation with reliable computing and design of a more general middleware framework are at a very early stage.

While we cannot demonstrate fault tolerance in our current experiments, we will briefly address the question of how the Transnet architecture enables it. The important point is that the entire state of our merge operation is exposed to the client through IBP. This means that any form of migration and replication of state can be implemented by the client simply by the execution of the appropriate sequence of IBP data movement operations, properly synchronized with instruction execution. Furthermore, the credit system provides sufficient control over multithreaded execution on the depot to enable even a very tight loop such as our merge example to be interrupted and state management operations inserted by the endpoint. All of this can be accomplished with minimal impact on performance through judicious use of the instruction cache.

The simplicity of our merge example derives in part from the fact that the merge loop takes no account of communication errors. It is also possible that such a simple loop would be used when operating in a highly reliable local area network, or using a reliable network or link layer connection. Alternatively, some applications might be robust to simply dropping data buffers where communication errors occur, and so the loop might be used in an environment where correctness criteria were very loose. However, if we assume that the loop is used in optimistic fashion when the error rate is very low, then a communication error might cause it to fail, and in that case the client might have to step in to replace this simple code with more complex code that, for instance, implemented acknowledgements and sliding windows on a per-hop basis. This more complex code would, undoubtedly, cause a performance penalty, and would be inserted only when necessary to maintain correct

operations. The evaluation of network conditions, and the management of the fault tolerant code, which might operate in a less autonomous mode (fewer credits) in order to allow action to be taken in response to the results of particular operations, would be the responsibility of the client.

5.2. Correctness

Fault tolerance through redundant management of state is adequate in a fail-stop environment, but truly weak semantics require that we make no assumptions about the behavior of a faulty intermediate node. This means that data movement, storage or processing can fail through by returning incorrect results, and even behaving maliciously. There are limits to the degree to which open networks can be protected from attacks such as Denial of Service, but security and correctness can be ensured to a high level of confidence in data storage and transfer services through the use of end-to-end checksums and encryption.

The application of techniques analogous to checksums and encryption to processing is a challenging field of research, and one that draws from a number of areas, including cryptography and peer-to-peer systems. It is possible in some cases to define operations specifically to allow fast verification of their results by the endpoint. Use of redundant computations performed at independent nodes is another way of checking accuracy. Ensuring privacy by processing data while it is encrypted is another interesting area of theoretical research.

For the immediate future, we intend to experiment with small-scale examples and to develop ad-hoc solutions to ensuring correctness while investigating more general approaches to the problem. However we recognize that no shared infrastructure can hope to survive for long today without a vigorous effort to at least detect and avoid being compromised by the most virulent methods of attack.

6. CONCLUSION AND RELATED WORK

In this paper, we have shown how pipelining and caching can be used to optimize the performance of the Internet Backplane Protocol, an overlay implementation of the Transnetworking architecture that offers exposed data transfer, storage and processing resources. Our experimental results clearly indicate that this approach can yield high performance in operations that require high utilization of intermediate node resources.

Related Work — Logistical Networking touches on most aspects of resource management for wide area distributed systems, and so it has an overlap with many areas of Distributed Operating System. The important categories are remote job execution [13], remote procedure call, state replication for fault tolerance and mobile code and agent infrastructures [7].

The area of Active Networks [10, 17, 26] 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.

The Logistical Session Layer (LSL) of Swany and Wolsky [25] is another approach to augmenting the functionality of the intermediate node which also creates encapsulated state through the instantiation of processes that maintain session state using the processing resources of conventional routers. Although the name might suggest otherwise, LSL is not implemented using IBP or any similar exposed approach, and so it represents an ad hoc extension to the stack, rather than an extensible framework such as Active Networking or an exposed platform like logistical networking based on IBP.

Calvert, Griffioen and Wen [8] 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 Networking: storage allocations are limited to a 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 [22, 24] 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 at 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. ACKNOWLEDGEMENT

The authors gratefully acknowledge the seminal contributions of James S. Plank to the original design of the Internet Backplane Protocol and his leadership in the definition of the LoCI Storage Stack and the development of the LoRS toolkit.

8. REFERENCES

[1] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris, "Resilient Overlay Networks," in *Procs. of the 18th ACM SOSP*. Banff, CAN, 2001.

[2] A. Bassi, et al., "The Internet Backplane Protocol: A Study in Resource Sharing," in *IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2002)*. Berlin, Germany: IEEE, 2002.

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

[4] M. Beck, Y. Ding, E. Fuentes, and S. Kancherla, "An Exposed Approach to Reliable Multicast in Heterogeneous Logistical Networks." In Workshop on Grids and Advanced Networks (GAN03), Tokyo, Japan, May 12-15, 2003.

[5] M. Beck, et al., "An End-to-End Approach to Globally Scalable Programmable Networking," *Journal of Computer Communications* Spring/Summer, 2004 (to appear).

[6] M. Beck, T. Moore, and J. S. Plank, "An End-to-end Approach to Globally Scalable Network Storage," in *Proceedings of SIGCOMM 2002*. Pittsburgh, PA, 2002, pp. 339-346.

[7] E. A. Brewer, et al., "A Network Architecture for Heterogeneous Mobile Computing," *IEEE Personal Communications Magazine*, vol. 5, no. 5, pp. 8-24, October, 1998.

[8] 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.

[9] Computer Science and Telecommunications Board (CSTB) of the National Research Council, "The Internet's Coming of Age." Washington, DC: National Academy Press, 2001, pp. 256.

[10] D. Decasper, G. Parulkar, and B. Plattner, "A Scalable, High Performance Active Network Node," *IEEE Network* January, 1999.

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

[12] K. Fall, "A Delay-Tolerant Network Architecture for Challenged Internets," in *Proceedings of SIGCOMM 2003*. Karlsruhe, DE: ACM, 2003.

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

[14] R. L. Grossman, et al., "Experimental Studies Using Photonic Data Services at IGrid 2002," *Journal of Future Computer Sys*, vol. 19, no. 6, pp. 945-955, 2003.

[15] R. A. Iannucci, "Two Fundamental Issues in Multiprocessing," in *Parallel Computing in Science and Engineering*, pp. 61-88, 1987.

[16] K. L. Johnson, J. F. Carr, M. S. Day, and M. F. Kaashoek, "The Measured Performance of Content Distribution Networks," in *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, 2000.

[17] R. Keller, L. Ruf, A. Guindehi, and Bernhard Plattner, "PromethOS: A Dynamically Extensible Router Architecture Supporting Explicit Routing." In Proc. of IWAN 2002, Zurich, Switzerland, December, 2002.

[18] T. Moors, "A critical review of 'End-to-end arguments in system design'." In Procceedings of the International Conference on Communications (ICC), New York, NY, Apr. 28 - May 2, 2002.

[19] J. S. Plank, et al., "Managing Data Storage in the Network," *IEEE Internet Computing*, vol. 5, no. 5, pp. 50-58, September/October, 2001.

[20] 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.

[21] S. Rhea, et al., "Maintenance-Free Global Data Storage," *IEEE Internet Computing*, vol. 5, no. 5, pp. 40-49, September/October, 2001.

[22] M. Ripeanu, I. Foster, and A. Iamnitch, "Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design," *IEEE Internet Computing Journal*, vol. 6, no. 1, 2002.

[23] 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.

[24] I. Stoica, et al., "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," in *SIGCOMM 2001*. San Diego, CA, 2001.

[25] D. M. Swany and R. Wolski, "Data Logistics in Network Computing: The Logistical Session Layer." In IEEE International Symposium on Network Computing and Applications (NCA'01), Cambridge, MA, USA, October, 2001.

[26] D. L. Tennenhouse, et al., "A Survey of Active Network Research," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80-86, January, 1997.