The Logistical Computing Stack –
A Design For Wide-Area, Scalable, Uninterruptible Computing

James S. Plank

Micah Beck

Department of Computer Science
University of Tennessee
Knoxville, TN 37996

[plank,mbeck]@cs.utk.edu
http://www.cs.utk.edu/~plank
http://loci.cs.utk.edu

A four-page paper giving an overview of our presentation
given at the Workshop on Scalable, Uninterruptible Computing,
part of DSN 2002: Dependable Systems and Networks,
Bethesda, Maryland, June, 2002

This was published by IEEE press as a supplement to the
conference proceedings

The Logistical Computing Stack – A Design For Wide-Area, Scalable, Uninterruptible Computing

James S. Plank Micah Beck*

Logistical Computing and Internetworking Lab
Department of Computer Science, University of Tennessee
Knoxville, TN 37996

Introduction

As the most recent wave of innovation in network computing crested near the end of the last decade, the enthusiasm for the next generation of information technologies was running very high. Inspired by the global impact of the Internet and the Web, and fueled by exponential increases in all basic computing resources, the network computing community spawned a range of ambitious new approaches and applications (e.g. active networks, mobile agents, ubiquitous computing, scientific collaboratories, etc.), many of which received significant development and had impressive early deployments. What seems evident today, however, is that this group of much anticipated innovations has had relatively little impact on advances in information technology practice; despite the rising flood of resources, the next wave of innovation has not yet materialized.

The concept of computation and storage residing *in the network*, instead of just being connected *by the network* is powerful. An in-the-network computing framework driven by this concept and supported by a highly programmable interface to individual application developers will liberate applications to scale to millions of instances that intelligently utilize a rich amount of computational, storage and network bandwidth resources.

The current manner in which processor and storage resources are managed in networked computing does not measure up to this challenge. The crux of this challenge, as we see it, is to enable users to benefit from provisioning of resources that are external not only the user's workstation but also to their immediate organization. This is an important part of the power of infrastructures such as the Internet and the Web: users have access to resources that they have no awareness of until they need to use them.

A guiding idea of our research program in Logistical Computing and Internetworking (LoCI), and the work on Logistical Networking at its core, is that the chief impediment to progress in this area is the failure to rethink and redesign the way in which storage and processor resources are integrated into the underlying "fabric layer" of resources [FKT01], the "programmatic substrate" [AMK98] on which these new technologies are trying to build. In this paper, we describe the co-design of the storage and processing elements of this fabric that will enable globally scalable and uninterruptible computing applications.

The Logistical Computing Stack

Our goal with the Logistical Computing Stack (Figure 1) is to layer abstractions of communication, storage and computation so that these resources may be part of the wide-area network in an efficient, flexible, sharable and scalable way. Its model, which achieves all these goals for data transmission, is the IP stack, and its guiding principle has been to follow the tenets laid out by End-to-End arguments [RSC98, SRC84]. Two fundamental principles of this layering are that each layer should (a) *abstract* the layers beneath it in a meaningful way, but (b) *expose* an appropriate amount of its own resources so that higher layers may abstract them meaningfully (see [BMP01] for more detail on this approach).

We omit the communication stack, as it is well-understood and documented elsewhere. We also omit discussion of the bottom two levels of the stack since they are obvious. Instead, we concentrate on the storage and computation stacks.

*This material is based upon work supported by the National Science Foundation under grants ACI-9876895, EIA-9975015, EIA-9972889, and Department of Energy under and the SciDAC/ASCR program, and the University of Tennessee Center for Information Technology Research.

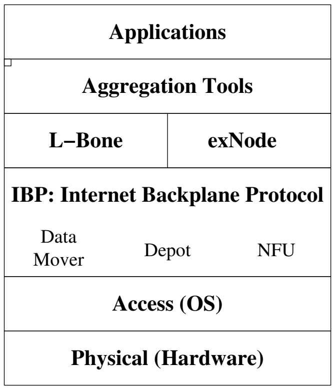


Figure 1: The Logistical Computing Stack

The Network Storage Stack

The IBP Depot: Again, the network storage stack has been previously explained and implemented, and its effectiveness has been demonstrated elsewhere [BMP02, PBB+01, ASP+02]. However, an explanation will help in the explanation of the computation stack. The lowest non-trivial layer of the storage stack is the Internet Backplane Protocol (IBP) storage depot. This is server daemon software and a client library that allows storage owners to insert their storage into the network, and to allow generic clients to allocate and use this storage. The unit of storage is a *time-limited, append-only byte-array*. These attributes are effective in allowing the depot to serve storage in a safe manner to untrusted, and even unknown clients. Byte-array allocation is like a network *malloc()* call – clients request an allocation from a specific IBP depot, and if successful, are returned trios of cryptographically secure text strings (called *capabilities*) for reading, writing and management. Capabilities may be used by any client in the network, and may be passed freely from client to client, much like a URL.

The IBP does its job as a low-level layer in the storage stack. It abstracts away many details of the underlying physical storage layers: block sizes, storage media, control software, etc. However, it also exposes many details of the underlying storage, such as network location, network transience and the ability to fail, so that these may be abstracted more effectively by higher layers in the stack.

The L-Bone and exNode: While individual IBP allocations may be employed directly by applications for some benefit [PBB+01], they, like IP datagrams, benefit from some higher-layer abstractions. The next layer contains the L-Bone, for resource discovery and proximity resolution, and the exNode, a data structure for aggregation.

The L-Bone (Logistical Backbone) is a distributed runtime layer that allows clients to perform IBP depot discovery. IBP depots register themselves with the L-Bone, and clients may then query the L-Bone for depots that have various characteristics, including minimum storage capacity and duration requirements, and basic proximity requirements. For example, clients may request an ordered list of depots that are close to a specified city, airport, US zipcode, or network host. Once the client has a list of IBP depots, it may then request that the L-Bone use the Network Weather Service (NWS) [WSH99] to order those depots according to bandwidth predictions using live networking data. Thus, while IBP gives clients access to remote storage resources, it has no features to aid the client in figuring out which storage resources to employ. The L-Bone's job is to provide clients with those features.

The exNode is a data structure for aggregation, analogous to the Unix inode (Figure 2). Whereas the inode aggregates disk blocks on a single disk volume to compose a file, the exNode aggregates IBP byte-arrays to compose a logical entity like a file. Two major differences between exNodes and inodes are that the IBP buffers may be of any size, and the extents may overlap and be replicated. In the present context, the key point about the design of the exNode is that it allows us to create 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 Network Stack approach.

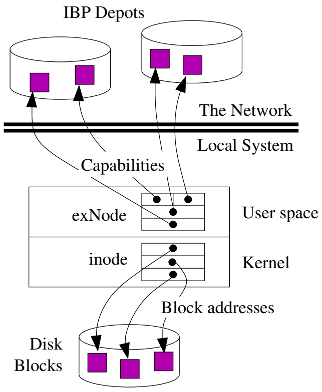


Figure 2: The exNode in comparison to the Unix inode

For maximal interoperability, exNodes are expressed concretely as an encoding of storage resources (typically IBP capabilities) and associated metadata in XML. Like IBP capabilities, these serializations may be passed from client to client, allowing a great degree of flexibility and sharing of network storage. If the exNode is placed in a directory, the file it implements can be embedded in a namespace. But if the exNode is sent as a mail attachment, there need not be a canonical location for it. The use of the exNode by varying applications provides interoperability similar to being attached to the same network file system.

Logistical (Aggregation) Tools: At the next level are tools that perform the actual aggregation of storage resources, using the lower layers of the Network Stack. These tools take the form of client libraries that perform basic functionalities, and standalone programs built on top of the libraries. Basic functionalities of these tools are listed below.

Upload: This takes local storage (e.g. a file, or memory), uploads it into the network and returns an exNode for the upload. This upload may be parameterized in a variety of ways. For example, the client may partition the storage into multiple blocks (i.e. stripe it) and these blocks may be replicated on multiple IBP servers for fault-tolerance and/or proximity reasons. Moreover, the user may specify proximity metrics for the upload, so the blocks have a certain network location.

Download: This takes an exNode as input, and downloads a specified region of the file that it represents into local storage. This involves coalescing the replicated fragments of the file, and must deal with the fact that some fragments may be closer to the client than others, and some may not be available (due to time limits, disk failures, and standard network failures). If desired, the download may operate in a streaming fashion, so that the client only has to consume small, discrete portions of the file at a time.

Refresh: This takes an exNode as input, and extends time limits of the IBP buffers that compose the file.

Augment: This takes an exNode as input, adds more replicas to it (or to parts of it), and returns an updated exNode. Like *upload*, these replicas may have a specified network proximity.

Trim: This takes an exNode, deletes specified fragments, and returns a new exNode. These fragments may be specified individually, or they may be specified to be those that represent expired IBP allocations. Thus, *augment* and *trim* may be combined to effect a routing of a file from one network location to another.

Applications: The end result for applications is that they may employ storage as a network resource as effectively as they employ communication – flexibly, scalably, safely, and an a manner resilient to failures. The code implementing the network storage stack is available for free download, and may be obtained along with documentation from the LoCI web page: `loci.cs.utk.edu`.

The Network Computation Stack

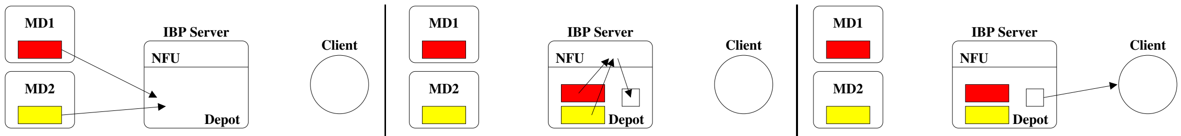
The Network Functional Unit (NFU): The key unit of network storage is the IBP depot allocation. Analogously, we define the key unit of network computation to be an 'atom' of computation on the IBP NFU. This atom of computation is a time-limited functional call for the IBP NFU to perform an operation on a list of allocations. These allocations *must* be local to the NFU's depot, and may be readable (input), and/or writable (output). The NFU executes the operation, the result of which may be normal termination, or suspension due to time-limit expiration. The operation is thus atomic. More importantly, the operation has two other properties:

1. It has no side effects other than writing to its output depot allocations.
2. It has no communication to other machines or to other processes on the host machine. This includes communications to the operating system of the host machine that are not part of primitive IBP operations.

The operations supported by a NFU are specific to the host IBP implementation. Example operations are operations implemented by functional RPC environments such as NetSolve [CD96] (e.g. linear algebra, scientific computations), graphics or multimedia operations, data mining operations, or virtual machine operations.

The first three of these examples are straighforward; however, a simple usage scenario will help illuminate. Suppose there are two databases of medical information, MD1 and MD2, and a client wishes to digest information from the two into a short summary. This can be performed in a three-step manner as illustrated in Figure 3. First, an IBP server is selected to perform the digesting. Ideally, this server should be close to the medical databases. Next, (step a), the information is delivered to the IBP depot. Then (step b), the IBP NFU is invoked to digest the information, and the result is stored at the IBP depot. Finally, (step c), the digest is delivered to the client.

Although this scenario is simplistic, the design of IBP and the NFU allows it to become arbitrarily complex. For example, suppose that the medical information to be digested is extremely large. Then the digesting may proceed in several steps, where each step stores a portion of the information at the depot, and then incrementally digests the information in single NFU operations. If the NFU becomes too slow or refuses to allow the client to perform the operation, then the client may move the information to another IBP server, which



(a) Data is sent to the IBP depot from the databases.

(b) The NFU performs its operation to digest the information

(c) The digest is delivered to the client

Figure 3: NFU usage scenario

can resume the computation. If desired, multiple NFU's may be employed to digest portions of the information in parallel (or in replication).

In a more complex vein, the operation performed by a NFU may be a time-slice of virtual machine execution. In such a case, the input allocations compose the state of the machine, and the outputs reflect the state transition of a time slice. It is conceivable that the state of computation may be scattered on IBP depots throughout the network, and the result of a NFU operation may be that it needs information from another depot. This is analogous to paging on a standard process on a single machine – the NFU's operation completes, but notes that it has incomplete information, and the computation may only continue on that NFU once the information is brought into the NFU's depot.

The L-Bone and the exNode: As in the network storage stack, the L-Bone may be used to perform NFU discovery, and the exNode may be used as a data structure for aggregation. However, instead of the aggregation being of just storage, it is of storage and computation. Examples are listed below.

Composition: Multiple NFU's may be composed to perform portions of a computation. This may be for reasons of parallelization, or perhaps because the computation portions are best performed close to the data.

Replication: NFU's may be instructed to replicate a computation or portions of a computation for purposes of fault-tolerance.

Migration: Since NFU operations are time-limited, the completion of an operation may yield merely an intermediate state of the computation. In other words, the operation's result may be viewed as a checkpoint of the computation, and as such, migration to another NFU is as simple as moving the relevant allocations to a new depot and starting a new operation.

Aggregation Tools: Moving higher, the main aggregation tool for the computation stack will be a *computation manager*. Like the logistical tools for storage, this manager will make the placement and scheduling decisions that employ the lower layers of the stack. This manager has not yet been designed. As with our research on the storage stack, our experience with direct programming of the lower levels will give us insight into how to design and implement the higher levels. However, it is important to note that the design philosophy has been the same: the components of the stack have been designed so that the higher levels have the ability to explicitly schedule each important component of the lower levels. Moreover, the IBP servers are afforded control in the form of time-limited storage and computation allocations, which allows them to be inserted safely into the network as a sharable resource. We believe that this design is the true path to wide-area, high-performance, scalable, uninterruptible computing.

References

[AMK98] E. Amir, S. McCanne, and R. Katz. An active service framework and its application to real-time multimedia transcoding. In *ACM SIGCOMM '98*, Vancouver, 1998.

[ASP+02] S. Atchley, S. Soltesz, J. S. Plank, M. Beck, and T. Moore. Fault-tolerance in the network storage stack. In *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, Ft. Lauderdale, FL, April 2002.

[BMP01] M. Beck, T. Moore, and J. S. Plank. Exposed vs. encapsulated approaches to grid service architecture. In *2nd International Workshop on Grid Computing*, Denver, 2001.

[BMP02] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable network storage. In *SIGCOMM 2002*, Pittsburgh, August 2002.

[CD96] H. Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. In *International Conference on Supercomputing*, 1996.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[PBB+01] J. S. Plank, A. Bassi, M. Beck, T. Moore, D. M. Swany, and R. Wolski. Managing data storage in the network. *IEEE Internet Computing*, 5(5):50–58, September/October 2001.

[RSC98] D. P. Reed, J. H. Saltzer, and D. D. Clark. Comment on active networking and end-to-end arguments. *IEEE Network*, 12(3):69–71, 1998.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems,*, 2(4):277–288, November 1984.

[WSH99] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5-6):757–768, 1999.