The Internet Backplane Protocol: Storage in the Network

James S. Plank Micah Beck Wael R. Elwasif Terence Moore Martin Swany Rich Wolski

Department of Computer Science
University of Tennessee
Knoxville, TN 37996

[plank,mbeck,elwasif,tmoore,swany,rich]@cs.utk.edu

Appearing in:
NetStore '99: Network Storage Symposium
Seattle, WA, October, 1999
http://dsi.internet2.edu/netstore99

http://www.cs.utk.edu/~plank/plank/papers/NS99-IBP.html

Abstract

For distributed and network applications, efficient management of program state is critical to performance and functionality. To support domain- and application-specific optimization of data movement, we have developed the *Internet Backplane Protocol* (IBP) for controlling storage that is implemented as part the network fabric itself. IBP allows an application to control intermediate data staging operations explicitly as data is communicated between processes. As such, the application can exploit locality and manage scarce buffer resources effectively. In this paper, we discuss the development of IBP, the implementation of a prototype system for managing network storage, and a preliminary deployment as part of the Internet-2 Distributed Storage Initiative.

1 Introduction

The proliferation of applications that are performance limited by network speeds leads us to explore new ways to exploit data locality in distributed settings. Currently, standard networking protocols (such as TCP/IP) support *end-to-end resource control*. An application can specify the endpoints associated with a communication stream, and possibly the buffering strategy to use at each endpoint, but has little control over how the communicated data is managed while it is traversing the network. In particular, it is not possible for the application to influence where and when data may be stored (other than at the endpoints) so that it can be accessed efficiently.

To support domain- and application-specific optimization of data movement, we have developed the *Internet Backplane Protocol* (IBP) for managing storage within the network fabric itself. IBP allows an application to implement interprocess communication in terms of intermediate data staging operations so that locality can be exploited and scarce buffer resources managed more effectively.

The design of IBP illustrates the tension between two different approaches to the design of large-scale distributed systems, the *functional* (or *behavioral*) model, which hides the internal state of each element, and the *shared-memory model*, which exposes a portion of that state. The advantage of the functional approach is that it is both robust and scalable. In this model, the failure of one element results only in the failure of current interactions and the future unavailability of that element, and the size of the system is limited only by the ability of elements to communicate. By contrast, the shared memory approach introduces long-lived dependencies between elements that can compromise both robustness and scalability. A failure that corrupts the shared state will be visible to any future reader, and the size of the system is limited by the ability to coordinate the sharing of state correctly. However, the shared-memory model does allow the application to have direct control of resources so that domain- and application-specific optimization of memory usage can be implemented.

In the case of the Internet, a large distributed system providing a communication service to systems at its periphery, such considerations of robustness and scalability led its designers to choose a stateless model, which is the networking equivalent of the functional model of computation. Routers perform stateless data transfer operations and control is calculated by routing algorithms which require only that the routers behave correctly, not that they maintain shared state. Indeed, the *end-to-end model* of IP networking has served the Internet well, allowing it to scale far beyond its original design while maintaining stable levels of performance and reliability. Why then would we seek to challenge this stateless approach?

To begin with, it is important to note that the designers of large-scale information systems often follow a shared-memory model because the functional model puts undesirable limitations on performance and control. It is difficult to express the management of stored data in functional terms, and resource management is key to high performance. Moreover, people think naturally in terms of processes, not just computations, and persistence is the key to defining a process. Increasingly the design of Internet-based information systems is moving toward shared-memory approaches that support the management of distributed data, rather than just access. A rapidly growing global industry is being built up around the caching and replication of content on the World Wide Web [8] and massive scientific datasets [2, 10, 20] are being distributed via networked systems of large storage resources.

But the management of shared state in such systems is in no way a part of the underlying network model. All such management is consequently pushed up to the application level and must be implemented using application-specific mechanisms that are rarely interoperable. The result has been a balkanization of state management capabilities that prevents applications that need to manage distributed state from benefiting from the kind of standardization, interoperability, and scalability that have made the Internet into such a powerful communication tool. We have defined the *Internet Backplane Protocol* (IBP) in order to provide a uniform interface to state management that is better integrated with the Internet. In this paper we will motivate and describe the IBP API and present some examples that show its strategic potential for the builders of distributed applications.

Because IBP is a compromise between two conflicting design models, it does not fit comfortably into either of the usual categories of mechanisms for state management: IBP can be viewed either as a mechanism to manage either *communication buffers* or *remote files* and both characterizations are equally valid and useful in different situations. If, in order to allow our terminology to be as neutral as possible, we simply refer to the units of data that IBP manages as *byte arrays*, then these different views of IBP can be presented as follows:

- **IBP as buffer management.** In communication between nodes on the Internet, which is built upon the basic operation of delivering packets from sender to receiver, each packet is *buffered* at each intermediate node. In other words, the communication makes use of storage in the network. Because the capacity of even large storage systems is tiny compared with the amount of data that flows through the Internet, allocation of communication buffers must be time limited. In current routers and switches, time-limited allocation is implemented by use of FIFO buffers, serviced under the constraints of fair queuing.

  Against this background, IBP byte arrays can be viewed as application-managed communication buffers in the network. IBP allows the use of time-limited allocation and FIFO disciplines to constrain the use of storage. These buffers can improve communication by way of application-driven staging of data, or they may support better network utilization by allowing applications to perform their own coarse-grained routing of data.

- **IBP as file management.** Since high-end Internet applications often transfer gigabytes of data, the systems to manage storage resources for such applications are often on the scale of gigabytes to terabytes in size. Storage on this scale is usually managed using highly structured file systems or databases with complex naming, protection and robustness semantics. Normally such storage resources are treated as part of a host system and therefore as more or less private.

  From this point of view IBP byte arrays can be viewed as files that live in the network. IBP allows an application to read and write data stored at remote sites, and direct the movement of data among storage sites and to receivers. In this way IBP creates a shareable network resource for storage in the same way that standard networks provide shareable bandwidth for file transfer.

This characterization of IBP as a mechanism for the management of state in the network supplies an operational understanding of the approach we are proposing, but it does not provide a unified view that synthesizes storage and networking together. In order to arrive at this more general view, we say that routing of packets through a network is a series of *spatial* choices that allows for control of only one aspect of data movement. An incoming packet is sent out on one of several alternative links, but any particular packet is held in communication buffers for as short a time as possible.

IBP, on the other hand, allows for data to be stored at one location while en route from sender to receiver, adding the ability to control data movement *temporally* as well as spatially. We term this generalized notion of data movement *logistical networking*, drawing an analogy with systems of warehouses and distribution channels commonly used in the logistics of transporting military and industrial material. Logistical networking can improve an application's performance by allowing files to be staged near where they will be used, data to be collected near its source, or content to be replicated close to its users. But to see how IBP implements this concept of logistical networking, we need to look at the API in detail.

2 IBP Structure and Client API

IBP has been designed to be a minimal abstraction of storage to serve the needs of logistical networking, i.e. to manage the path of data through both time and space. The fundamental operations are:

1. Allocating a byte array for storing data.
2. Moving data from a sender to a byte array.
3. Delivering data from a byte array to a receiver (either another byte array or a client).

We have defined and implemented a client API for IBP that consists of seven procedure calls, and server daemon software that makes local storage available for remote management. Currently, connections between clients and servers are made through TCP/IP sockets. As we experiment with IBP, we plan to explore other networking protocols (e.g. UDP) that can move IBP functionality closer to the network.

IBP client calls may be made by *anyone* who can attach to an IBP server (which we also call an *IBP depot* to emphasize its logistical functionality). IBP depots require only storage and networking resources, and running one does not necessarily require supervisory privileges. These servers implement policies that allow an initiating user some control over how IBP makes use of storage. An IBP server may be restricted to use only idle physical memory and disk resources, or to enforce a time-limit on all allocations, ensuring that the host machine is either not impacted at all, or only impacted for a finite duration. The goal of these policies is to encourage users to experiment with logistical networking without over-committing server resources.

Logically speaking, the IBP client sees a depot's storage resources as a collection of append-only byte arrays. There are no directory structures or client-assigned file names. Clients initially gain access to byte arrays by allocating storage on an IBP server. If the allocation is successful, the server returns three *capabilities* to the client: one for reading, one for writing, and one for management. These capabilities can be viewed as names that are assigned by the server. Currently, each capability is a text string encoded with the IP identity of the IBP server, plus other information to be interpreted only by the server. This approach enables applications to pass IBP capabilities among themselves without registering these operations with IBP, and in this way supports high-performance without sacrificing the correctness of applications.

The IBP client API consists of seven procedure calls, broken into three groups, defined in Figure 1. We omit error handling for clarity. The full API is described in a separate document [18].

2.1 Allocation

The heart of IBP's innovative storage model is its approach to allocation. Storage resources that are part of the network, as logistical networking intends them to be, cannot be allocated in the same way as they are on a host system. Any network is a collection of *shared resources* that are allocated among the members of a highly distributed community. A public network serves a community which is not closely affiliated and which may have no social or economic basis for cooperation. To understand how IBP needs to treat allocation of storage for the purposes of logistical networking, it is helpful to consider the problem of sharing resources in the Internet, and how that situation compares with the allocation of storage resources on host systems.

In the Internet, the basic shared resources are data transmission and routing. The greatest impediment to sharing these resources is the risk that their owners will be denied the use of them. The reason that the Internet can function in the face of the possibility of denial-of-use attacks is that it is not possible for the attacker to profit in proportion to their own effort, expense and risk. When other resources, such as disk space in spool directories, are shared, we tend to find administrative mechanisms that limit their use by restricting either the size of allocations or the amount of time for which data will be held.

By contrast, a user of a host storage system is usually an authenticated member of some community that has the right to allocate certain resources and to use them indefinitely. Consequently, sharing of resources allocated in this way cannot extend to an arbitrary community. For example, an anonymous FTP server with open write permissions

Allocation:

```
IBP_cap_set IBP_allocate(char *host, int size, IBP_attributes attr)
```

Reading / Writing:

```
IBP_store(IBP_cap write_cap, char *data, int size)
IBP_remote_store(IBP_cap write_cap, char *host, int port, int size)
IBP_read(IBP_cap read_cap, char *buf, int size, int offset)
IBP_deliver(IBP_cap read_cap, char *host, int port, int size, int offset)
IBP_copy(IBP_cap source, IBP_cap target, int size, int offset)
```

Management:

```
IBP_manage(IBP_cap manage_cap, int cmd, int capType, IBP_status info)
```

Figure 1: The IBP Client API

is an invitation for someone to monopolize those resources; such servers must be allowed to delete stored material at will.

In order to make it possible to treat storage as a shared network resource, IBP supports some of these administrative limits on allocation, while at the same time seeking to provide guarantees that are as strong as possible for the client. So, for example, under IBP allocation can be restricted to a certain length of time, or specified in a way that permits the server to revoke the allocation at will. Clients who want to find the maximum resources available to them must choose the weakest form of allocation that their application can use.

To allocate storage at a remote IBP depot, the client calls **IBP_allocate()**. The maximum storage requirements of the byte array are noted in the **size** parameter, and additional attributes (described below) are included in the **attr** parameter. If the allocation is successful, a trio of capabilities is returned.

There are several allocation attributes that the client can specify:

- **Permanent vs. time-limited**. The client can specify whether the storage is intended to live forever, or whether the server should delete it after a certain period of time.
- **Volatile vs. stable**. The client can specify whether the server may revoke the storage at any time (volatile) or whether the server must maintain the storage for the lifetime of the buffer.
- **Byte-array/Pipe/Circular-queue**. The client can specify whether the storage is to be accessed as an append-only byte array, as a FIFO pipe (read one end, write to another), or as a circular queue where writes to one end push data off of the other end once a certain queue length has been attained.

We anticipate that applications making use of shared network storage, (storage that they do not explicitly own), will be constrained to allocate either permanent and volatile storage, or time-limited and stable storage.

2.2 Reading / Writing

All reading and writing to IBP byte arrays is done through the four reading/writing calls in Figure 1.

These calls allow clients to read from and write to IBP buffers. **IBP_store()** and **IBP_read()** allow clients to write from and read to their own memory. **IBP_remote_store()** and **IBP_deliver()** allow clients to direct an IBP server to connect to a third party (via a socket) for writing/reading. Finally, **IBP_copy()** allows a client to copy an IBP buffer from one server to another. Note that **IBP_remote_store()**, **IBP_deliver()** and **IBP_copy()** all allow a client to direct an interaction between two other remote entities. The support that these three calls provide for third party transfers are an important part of what makes IBP different from, for example, typical distributed file systems.

The semantics of **IBP_store()**, **IBP_remote_store()**, and **IBP_copy()** are append-only. **IBP_read()**, **IBP_deliver()** and **IBP_copy()** allow portions of IBP buffers to be read by the client or third party. If an IBP server has removed a buffer, due to a time-limit expiration or volatility, these client calls simply fail, encoding the reason for failure in an **IBP_errno variable**.

2.3 Management / Monitoring

All management of IBP byte arrays is performed through the **IBP_manage()** call. This procedure requires the client to pass the management capability returned from the initial **IBP_allocate()** call. With **IBP_manage()**, the client may manipulate a server-controlled reference count of the read and write capabilities. When the reference count of a byte array's write capability reaches zero, the byte array becomes read-only. When the read capability reaches zero, the byte array is deleted. Note that the byte array may also be removed by the server, due to time-limited allocation or volatility. In this case, the server invalidates all of the byte array's capabilities.

The client may also use **IBP_manage()** to probe the state of a byte array and its IBP server, to modify the time limit on time-limited allocations, and to modify the maximum size of a byte array.

3 Application Strategies with Logistical Networking using IBP

For application developers, logistical networking means having the power to manage the temporal aspects of the trajectory that data takes through the network. IBP provides explicit control over these characteristics through the use of capabilities. While this control makes possible a range of logistical networking strategies that is seemingly limitless, it will take time and experience to determine which approaches yield the greatest improvements in application performance, application functionality, or overall resource utilization. To provide some preliminary idea of the potential value of IBP, however, below we present a few straightforward strategies for logistical networking and describe some representative applications that can make use of them.

3.1 Storing Data Near the Sender: Memory Servers for Distributed Sensor Data and Checkpointing

When transmission of data across a wide area link is slow, a nearby IBP-enabled storage server can act as a surrogate receiver, freeing the sender from the need to buffer the data. This strategy can also be used to implement lazy data transmission when wide area transfer may be unnecessary.

For instance, the Network Weather Service (NWS) is a system that monitors network resources and predicts their future behavior [34]. It is one of many applications that manage and collect distributed sensor data. Performance data is periodically gathered from a distributed set of "sensors" and kept in persistent storage for later processing. When a prediction of future performance is required (e.g. at scheduling time) the NWS applies a set of numerical forecasting models to the most recent performance data and makes a prediction of available resource response. The monitoring part of NWS typically stores data near the sender of monitoring information since performance measurement is usually much more frequent than forecast generation. Moreover, the dynamically changing performance response of networks, CPUs, and memory systems makes old data obsolete. The NWS persistent storage servers are structured like the circular queues of IBP — the monitors append data to the end of the queue, pushing data off the front of the queue if the queue has reached a prespecified size. When a NWS client needs to make a forecast, the system reads a limited history of the monitoring data held in the relevant persistent storage servers and applies the forecasting models. The result is passed to the client via an explicit message.

Currently, the persistent state servers used by NWS are special-purpose and NWS-specific, which makes deploying the system more difficult. The plan for the next revision of NWS includes employing IBP buffers as the persistent state mechanism. In part this is being done to simplify the code of NWS, and in part to leverage the deployment momentum of IBP (via I2-DSI as described in Sec 4). Moreover, using IBP will make it possible to experiment with and implement different replication and consistency strategies more easily than with the current implementation.

Of course the collection of distributed sensor data is one instance of a more general problem: the collection and maintenance of continuously produced data from various distributed sources. Most distributed data management facilities (e.g. LDAP [37]) are implemented to optimize the performance of queries and not the performance of updates. For facilities gathering dynamic performance data, however, the speed of updates is critical and data lifetime is limited.

A similar type of application that can benefit from the presence of distributed, IBP-enabled memory servers is *checkpointing*. Checkpointing — the saving of program state to some external storage medium — is the most successful mechanism for fault-tolerance in all areas of computing. Typically checkpoints are stored on disks on a local area network [1, 16, 24]. However, for reasons of performance, migratability, availability and extra fault-tolerance, it is often useful to employ a more flexible storage medium [27]. IBP is a natural facility for managing this storage, providing most of the necessary primitives for this task. As checkpointing is added to core parts of the NetSolve server software library (see section 3.4), IBP will be used to manage the storage [25]. In this way, the location of the checkpoints can be managed by the client or agent in a manner that is independent of the checkpointing mechanism and server software. This in turn should facilitate process migration and scheduling in addition to fault-tolerance.

3.2 Storing Data Near the Receiver: IBP Mail and Speculative HTTP

One strategy for improving the delivery of large byte arrays from a sender to a receiver is to stage the data close to the receiver in advance, allowing the storage server to act as a surrogate sender. IBP is tailor-made for such an approach. A relatively simple but powerful example is provided by the way IBP can be used to enhance the familiar scheme by which digital objects are attached to mail messages.

The standard implementation of MIME-encoded attachments incorporates the encoded object into the SMTP stream flowing from the sender to the receiver. Because mail is an asynchronous mechanism, and because it must pass through various gateways and firewalls, the mail message and its attachments must be stored in "spool areas" at various points along its path. Because these spool areas are owned by some particular host, the amount of storage dedicated to spool areas is usually limited. Consequently, so is the size and sometimes the duration of any particular mail message.

Despite the evident success of MIME attachments, the current scheme is encumbered by various problems, not least of which is the fact that very large digital objects usually cannot be attached to e-mail messages. Video files and scientific data sets are still routinely sent on magnetic tape rather than over the network, even when network capacity allows it. Moreover, the need to move all kinds of large objects on a routine basis is growing rapidly. However, unless additional storage resources are made publicly available by another protocol (e.g FTP), the Internet does not provide resources for asynchronously transmitting such large objects. Moreover, existing protocols and administrative mechanisms make it difficult for storage to be securely and flexibly used from arbitrary points on the network. Finally, if a mail message with a large attachment is sent to a number of recipients, the problem is multiplied by the number of paths along which the mail must flow.

If the system is enhanced with IBP, however, the sender can simply store the file in an IBP buffer and include the IBP capability for accessing it in the mail message. Upon receiving the mail, the recipient downloads the file from the network. In the meantime, the file may be copied to an IBP buffer close to the receiver. IBP Mail has been implemented and is described in detail in a separate document [17].

Speculative transfers present a similar kind of optimization opportunity for IBP. Speculative transfers can be used to improve throughput in any information processing system where the latency of data transfer is much greater than the time it takes to process that data. In the case of the World Wide Web, for instance, speculative HTTP transfers involve moving objects that have not yet been requested. Attempts have been made to implement this approach by using a client-side cache making anticipatory transfers. But while this approach can be effective, it has failed to gain acceptance for two main reasons: it increases the load on already busy Web servers and Internet links, and it burdens the local resources of the client. An IBP-based implementation of the same strategy would allow the server to retain control of the process, transferring speculatively only when it is otherwise idle. It would also allow the server to transparently include objects stored in geographically dispersed depots into a given transfer. Finally, if speculative transfers targeted local IBP depots rather than the client's storage, there would be no impact on the client's own resources.

In fact, a system for using shared storage in the network can allow more sophisticated management of state in a session of *any* protocol that involves predictable future transfers. Our plan to develop a *Logistical Session Layer* protocol, described below, is intended to show that this is so.

3.3 Buffering Data at Network Boundaries: The Logistical Session Layer

When sending data directly from a sender to a receiver, it is sometimes possible to identify locations where, due to mismatches in transmission characteristics across network boundaries, it would be advantageous to increase the buffering of the data in FIFO fashion. This allows the storage server to act as *both* surrogate sender and surrogate receiver, giving rise to new optimizations of familiar and typical uses of networking.

We call one version of this strategy that we intend to explore the *Logistical Session Layer* (LSL). LSL is an application of logistical networking designed to address problems of bulk data transfer over networks with high bandwidth/delay products. Despite the many advances in TCP that have been made to address this situation, there is still a fundamental cost associated with buffering unacknowledged segments for retransmission. Moreover it is clear that the problem is only exacerbated as network speeds increase. For instance, sufficient buffering to support a TCP stream that half fills an OC-48 link from the east to west coast must be on the order of 15 megabytes, likely much higher. Following an approach which is similar to that proposed by Salehi *et al* for use in multicast video transmission [29], LSL will make use of IBP depots to insure that a packet loss need not require a retransmit from the original source, but rather may do so from an intermediate location.

LSL will be a “session” layer (layer 5) in terms of the OSI protocol model. Implementation of this architecture has the benefit of utilizing existing protocols beneath it. A connection that is initiated through the LSL can pass through a number of IBP depots. In this scenario these depots can actually be thought of as ”transport layer routers.” Recall that a transport layer conversation consists of multiple hops of network layer conversations. In an analogous fashion, we envision a session layer conversation to consist of multiple hops of IBP data transfers.

The LSL interface will closely mimic the Berkeley socket interface. In this way, existing programs that might benefit from this intermediate buffering can be easily modified to take advantage of it. These logistical sockets will initially be half duplex, and both the source and destination port of such a half-duplex connection will be assigned by the initiator. This will necessitate that the **lsl_bind()** on the recieve side be performed without specifying a local port.

When an LSL connection is initiated, a predicted path may be specified or local forwarding decisions may be relied upon. To specify a path explicitly, the sender will use the strict source route options with the LSL socket. In fact a combination of local and global forwarding strategies may be employed by specifying a loose source route in the same fashion.

3.4 Optimizing Producer/Consumer Transfers: State Management in NetSolve

When implementing distributed computation in a wide area network, data can be produced at any location in the network and consumed at any other, perhaps after a significant delay. Part of the difficult job of scheduling remote computation is making an intelligent choice of location for the producer, the consumer, and possibly for the buffer needed to connect them. High performance requires that both producer and consumer be kept close to the data whenever possible, and fulfilling this requirement can involve complex strategies for maintaining copies.

We are using NetSolve [11] as an experimental environment for exploring the potential of logistical networking for addressing this common problem. NetSolve is a software environment for networked computing that is currently in use at many institutions. Its design allows *clients* to access computational servers across the network using a familiar procedure call interface from a variety of programming languages and computational tools (e.g. Matlab). NetSolve uses a *client-agent-server* paradigm as de-

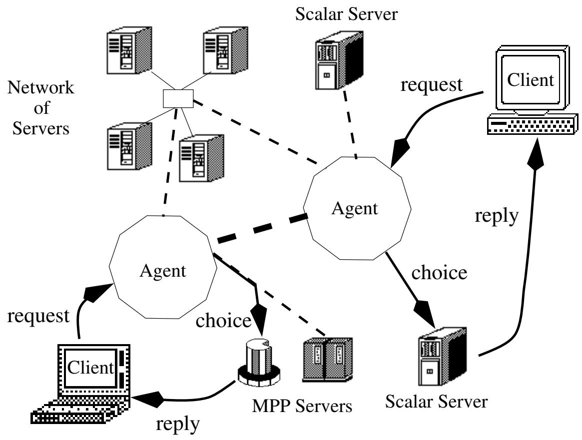

Figure 2: Basic structure of NetSolve

picted in Figure 2. NetSolve users are the *clients*, and the computational resources are the *servers*. A server may be a uniprocessor, a MPP (Massively Parallel Processor), or a networked cluster of machines. When a user wants a certain computational task to be performed, he/she contacts an *agent* with the request. Each agent maintains information such as availability, load, and supported software, on a collection of servers. When a request from a user comes in, the agent selects a server to perform the task, and the server responds to the client's request.

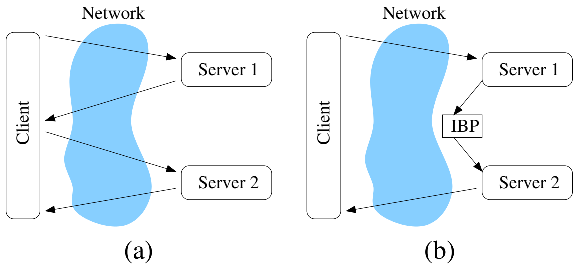

Figure 3: Scheduling consecutive NetSolve calls with a common argument. (a) No state management, (b) Storing temporary results in IBP near the computational servers.

NetSolve's computation model is functional: servers receive arguments from the clients, and then return results to the client. We are working with Dongarra (at the University of Tennessee), and Berman and Casanova (at the University of California, San Diego) to augment the NetSolve model to allow values to be stored in IBP depots and to then co-schedule storage and computation. As a simple example, consider a result produced by a call to a server and returned to the client, only to be sent as an argument to a subsequent call (Figure 3(a)). If the two calls can be scheduled on a single server or two nearby servers in the proximity of an IBP depot, the value that passes between them can be stored at the depot and wide area data transfer can be reduced (Figure 3(b)).

This example of two consecutive NetSolve calls is very simple. In general, the problem of state management in NetSolve may be represented by expressing the relevant calls in a computational graph, and then using the graph to schedule the computation and data movement. There are significant complexities in both expressing computations as dataflow or dependence flow graphs, and scheduling them in complex and changing environments.

There is a vast body of work on scheduling distributed computations [5,6], and scheduling computations based on dataflow graphs [30,35,36] and dependence flow graphs [3, 28] which we intend to use as the basis for NetSolve co-scheduling strategies. These range from dynamic strategies based on caching algorithms to allocation schemes based on graph partitioning. All such approaches exploit some ability to use and manage state in the network, and thus are relevant applications for IBP. Since NetSolve applications exist in forms that can be reduced to computation graphs, NetSolve will provide an excellent proving ground of IBP on real-world applications.

4 The L-Bone: Cooperative Logistical Networking

While data communication facilities are found in all information systems, internetworking has been especially important in creating a common infrastructure for diverse applications throughout the world to communicate with one another. Rather than connecting one application at a specific site to another application at another site, the Internet connects all IP applications at any connected site to all applications at any other connected site. Applications that use IBP are no different. While it is possible to provision the network with sufficient storage depots to meet the specific needs of communication between two endpoints, we believe that the power of logistical networking can only really be felt throughout the network when depots are widely deployed and storage can routinely be allocated in any part of the network were it is needed. We are now building an infrastructure to put this belief into effect.

The *Logistical Backbone* (or *L-Bone*) is a collection of IBP servers deployed in the Internet for the specific purpose of offering network storage to applications. The L-Bone will initially make use of some of the resources of the servers being deployed by the Internet2 Distributed Storage Infrastructure (I2-DSI) project [4].[1] These currently consist of five servers donated by IBM with 72GB of disk storage and 900GB of tape storage each and one server donated by StorageTek with 700GB of disk. Each site runs an IBP server that offers at least 20 gigabytes of storage for time-limited and/or volatile allocation. Additional systems will be added by I2-DSI, the University of Tennessee and other IBP project participants. Note that this storage capacity, if properly managed, can be used to augment endpoint storage that may be in short supply. In other words I2-DSI resources can be temporarily committed to overcome a shortfall at either endpoint of a desired communication.

The IBP server software has been designed so that any machine can become an IBP depot, serving up limited quantities of memory and/or disk storage, as determined by the machine owner, with the restrictions of time-limited or volatile allocation. The intent of this design is to encourage machine owners to donate the spare resources of their machines to the L-Bone. It is well known that most computers are not consistently or continuously utilized, and this has led to the successful implementations of cycle-stealing programming environments such as Condor [32] and Cosmic [14]. Our intent is for the L-Bone to be composed of dedicated storage resources, such as those allocated on the I2-DSI deployment machines, and contributed resources from individual and institutional machine owners. This mirrors the design of the academic Internet, with shared backbone links and peripheral links contributed by individual institutions. For more information on L-BONE participation, see `http://icl.cs.utk.edu/ibp`.

5 Related Work

Metacomputing systems are those that tie together large numbers of heterogeneous processing elements, perhaps across a wide area, in order to achieve high performance. Examples are Globus [19] and Legion [23]. Part of the Globus system is a storage management system called GASS [7], which manages movement of input and output files from system to system, as well as pre-fetching of input files and write-back of output files. More recent developments in Globus storage management are detailed in research on the Data Grid [12]. A separate project is an *active meta-data management system* [13] from Northwestern University, where programs access and manage distributed data with the help of an agent. This can be viewed as the NetSolve methodology applied to storage.

All of the above projects require remote storage resources to be managed and accessed in much the same

[1] I2-DSI is a joint project between Internet2 (a project of the University Corporation for Advanced Internet Development), the University of Tennessee, Knoxville and the University of North Carolina at Chapel Hill

way as IBP. We are exploring the use of IBP as underlying middleware for these projects, thus broadening their resource bases, and likely increasing portability. This endeavor should also give us insight into ways in which the IBP interface needs to be modified.

IBP draws to a certain extent on the experiences in the field of networking through its history. Store and forward connectivity has been used for quite some time in the networking community and there are many situations in which data transfer need not be synchronous or connection-oriented. SMTP [15,26], USENET [21] and its successor NNTP [22] all use hop-oriented, connectionless paradigms to achieve their goal.

The end-to-end model holds that all network state must be contained in the end nodes. However, even this has once again become an open issue in light of recent differentiated services work [9]. Further, the need to impose arbitrary network topologies over physical ones has been identified [33].

As a management system for network buffers, IBP is closely related to the Detour Project [31] underway at the University of Washington. It's primary focus has been to study the effects of "intelligent" packet routing strategies to improve network performance. IBP is certain to benefit from Detour results as they become available.

6 Conclusion

The Logistical Networking project seeks to expand the resources shared by applications communicating over the Internet to include storage as well as transmission of data. This approach opens up the temporal characteristics of the trajectory of data through the network for control. IBP has been designed as a minimal abstraction of storage in the network which we are using to experiment in a number of application areas. In order to experiment with IBP applications in a wide area setting, it is necessary to devote storage resources widely in the Internet. The L-Bone is a project that seeks to deploy IBP servers widely enough to create a realistic testbed in at least some parts of the network.

Assuming that applications see the anticipated benefits of logistical networking, the L-Bone is expected to grow in voluntary and cooperative manner so that applications that need storage close to a particular site can get it consistently via time-limited or volatile allocation from the L-Bone. If successful, the L-Bone and IBP will have a significant impact on the way that applications view and make use of the Internet. Instead of being a public data transmission network connecting private storage elements, it will be viewed as a public communication *and storage* network.

7 Acknowledgements

This material is based upon work supported by the National Science Foundation under grants EIA-9975015, ACI-9876895 and CCR-9703390, and by the Department of Energy under grant DE-FC0299ER25396.

References

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *8th IEEE International Symposium on High Performance Distributed Computing*, 1999.

[2] S. Baru. Storage Resource Broker (SRB) reference manual. `http://www.npaci.edu/DICE/SRB/SRB1_0/index.html`, 1997.

[3] M. Beck, R. Johnson, and K. Pingali. From control flow to dataflow. *Journal of Parallel and Distributed Computing*, 12:118–129, 1991.

[4] M. Beck and T. Moore. The Internet2 Distributed Storage Infrastructure project: An architecture for internet content channels. *Computer Networking and ISDN Systems*, 30(22-23):2141–2148, 1998.

[5] F. Berman. High-performance scheduling. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, July 1998.

[6] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing '96*, November 1996.

[7] J. Bester, I. Foster, K. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. In *Sixth Workshop on I/O in Parallel and Distributed Systems*, May 1999.

[8] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. Schwartz. The Harvest information and discovery system. *Computer Networks and ISDN Systems*, 28:119–125, 1995.

[9] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview. RFC 1633 (`http://www.ietf.org/rfc/rfc1633.txt`), June 1994.

[10] J. J. Bunn, H. B. Newman, and R. P. Wilkinson. Status report from the Caltech/CERN/HP "GIOD" joint project – globally interconnected object databases. In *CHEP '98: Computing in High Energy Physics*, Chicago, August 1998.

[11] H. Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.

[12] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets. In *NetStore '99: Network Storage Symposium*. Internet2, October 1999.

[13] A. Choudhary, M. Kandemir, H. Nagesh, J. No, X. Shen, V. Taylor, S. More, and R. Thakur. Data management for large-scale scientific computations in high performance distributed systems. In *Eighth IEEE International Symposium on High Performance Distributed Computing*, August 1999.

[14] P. E. Chung, Y. Huang, S. Yajnik, G. Fowler, K. P. Vo, and Y. M. Wang. Checkpointing in CosMiC: a user-level process migration environment. In *Pacific Rim International Symposium on Fault-Tolerant Systems*, December 1997.

[15] D. Crocker. Standard for the format of ARPA Internet text messages. IETF RFC 822 (`http://www.ietf.org/rfc/rfc0822.txt`), August 1982.

[16] E. N. Elnozahy, L. Alvisi, Y-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, June 1999.

[17] W. Elwasif, J. S. Plank, M. Beck, and R. Wolski. *IBP-Mail*: Controlled delivery of large mail files. In *NetStore '99: Network Storage Symposium*. Internet2, October 1999.

[18] W. R. Elwasif, J. S. Plank, and M. Beck. IBP - Internet Backplane Protocol: Infrastructure for distributed storage (V 0.2). Technical Report CS-99-430, University of Tennessee, February 1999.

[19] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, Summer 1998.

[20] R. Fox, S. Bates, and C. Jacobs. Unidata: A virtual community sharing resources via technological infrastructure. *Bulletin of the American Meteorological Society*, 78:457–468, March 1997.

[21] M. Horton. Standard for interchange of USENET messages. IETF RFC 850 (`http://www.ietf.org/rfc/rfc0850.txt`), June 1983.

[22] B. Kantor and P. Lapsley. A proposed standard for the stream-based transmission of news. IETF RFC 977 (`http://www.ietf.org/rfc/rfc0977.txt`), February 1986.

[23] M. J. Lewis and A. Grimshaw. The core Legion object model. In *Fifth IEEE International Symposium on High Performance Distributed Computing*, 1996.

[24] J. S. Plank. Program diagnostics. In John G. Webster, editor, *Wiley Encyclopedia of Electrical and Electronics Engineering*, volume 17, pages 300–310. John Wiley & Sons, Inc., New York, 1999.

[25] J. S. Plank, H. Casanova, M. Beck, and J. J. Dongarra. Deploying fault tolerance and task migration with NetSolve. *Future Generation Computer Systems*, 15:745–755, 1999.

[26] J. B. Postel. Simple Mail Transfer Protocol – SMTP. IETF RFC 821 (`http://www.ietf.org/rfc/rfc0821.txt`), August 1992.

[27] J. Pruyne and M. Livny. Managing checkpoints for parallel programs. In *Workshop on Job Scheduling Strategies for Parallel Processing (IPPS '96)*, 1996.

[28] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A convex programming approach for exploiting data and functional parallelism on distributed memory multicomputers. In *International Conference on Parallel Processing*, pages 116–125, 1994.

[29] J. D. Salehi, Z-L. Zhang, J. F. Kurose, and D. Towsley. Supporting stored video: Reducing rate variability and end-to-end resource requirements through optimal smoothing. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 222–231, Philadelphia, May 1996.

[30] V. Sarkar. *Partitioning and Scheduling of Parallel Programs for Multiprocessors*. MIT Press, 1989.

[31] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The end-to-end effects of Internet path selection. In *ACM SIGCOMM '99*, September 1999.

[32] T. Tannenbaum and M. Litzkow. The Condor distributed processing system. *Dr. Dobb's Journal*, #227:40–48, February 1995.

[33] J. Touch and S. Hotz. The X-Bone. In *Third Global Internet Mini-Conference, in conjunction with Globecom '98*, Sydney, Australia, November 1998.

[34] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15, 1999.

[35] R. M. Wolski. *Program Partitioning for NUMA Computer Architectures*. PhD thesis, University of California, Davis, 1994.

[36] T. Yand and A. Gerasoulis. A fast static scheduling algorithm for dags on an unbounded number of processors. In *Supercomputing '91*, November 1991.

[37] W. Yeong, T. Howes, and S. Kille. Lightweight directory access protocol. IETF RFC 1777 (`http://www.ietf.org/rfc/rfc1777.txt`), March 1995.