To the Graduate Council:
I am submitting herewith a dissertation written by Wael Rashad Elwasif entitled *The Internet Backplane Protocol: shared storage for enhanced network infrastructure*. I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

James S. Plank
Major Professor

We have read this dissertation
and recommend its acceptance:

Bradley Vander Zanden

Micah Beck

Robert Harrison

Accepted for the Council:

Anne Mayhew
Vice Chancellor
and Dean of the Graduate School

(Original signatures are on file with official student records)

The Internet Backplane Protocol: shared storage for enhanced network infrastructure

A Dissertation

Presented for the

Doctor of Philosophy Degree

The University of Tennessee, Knoxville

Wael Rashad Elwasif

May 2004

Acknowledgments

There are many people to whom I owe a great deal of gratitude and without whose help and support this dissertation could not have been completed. I would like to thank my dissertation advisor, Dr. Jim Plank, for his guidance during the course of my PhD studies, for the great courses that formed my view of the field of computer science, and for his support and insights during the course of this work. I would like to thank Dr. Micah Beck, who co-developed the original vision of IBP and logistical networking with Dr. Plank, for many insights and thought provoking discussions (and questions) along the course of this work. I'd like to thank Dr. Brad Vander Zanden and Dr. Robert Harrison for serving on my PhD committee, and for their thought provoking questions. I would like to also thank Dr. Rich Wolski for his help and invaluable input about different aspects of IBP and its applications.

I could not have finished this work without the support and encouragement of my dear wife, Samar, who kept me going forward through the long and hard process of wrapping up this work. My daughter Mariam provided more motivation (and some late night inspiration) than she could possibly imagine. My dear mother Dorria and my sister Dalia provide much needed encouragement and support that kept me going, thank you.

Abstract

Presented here are results pertaining to the utilization of distributed shared storage infrastructure to facilitate application management of remote state. The proposed infrastructure relies on non-privileged storage servers deployed by users at various location in the network. Applications utilize network topology information and storage servers profiles to develop and deploy data access and movement policies that enhance application performance and/or functionality. The combined management of storage and communication resources afforded by the proposed model enable a wide array of new applications that depart from the end-to-end model at the core of the existing Internet. However, this departure is done at a high enough level such that it doesn't compromise the advantages of the end-to-end model at lower levels in the communication protocol stack that are essential for network scalability. The proposed model enables locality-aware applications to improve overall resource utilization through execution policies that take into consideration data locality, storage availability at various locations, and communication bandwidth available. The major result of this dissertation is to demonstrate that the combined management of remote network storage and communication bandwidth information enhances the functionality and/or performance of certain classes of distributed network applications.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The recent explosive growth in the global Internet has been accompanied by the need to develop and deploy protocols and tools that allows shared access to various resources connected to the network. The notion of the network as an enabling technology that makes access to remote compute and storage resources possible lies at the core of the plethora of protocols that define the current state of the Internet. At the core of many of these protocols lies the notion of end-to-end resource control [33]. In this paradigm, state information necessary to the establishment and maintenance of a communication stream is maintained *only* at the two end hosts involved in the communication.

The end-to-end model has been fundamental in achieving the level of

scalability and fault tolerance observable in the global Internet. However, this same model prevents applications running on the end hosts from exploiting knowledge about the network topology and real-time characteristics to influence data movement through the network in a manner favorable to the specific needs of the application. While it can be safely argued that such management at the packet level is impractical, or even impossible, for most applications, doing so at a higher level in the communication protocol stack can be both feasible and beneficial to several applications categories.

This dissertation explores the issue of developing a middleware layer that allows applications to manage state remotely in the network. This layer, called the **I**nternet **B**ackplane **P**rotocol (IBP), allows the treatment of remote storage as a resource that can be accessed and shared by applications connected to the network. The dissertation further explores the combined management of remote storage and bandwidth as a means to enhance applications' performance and/or functionality. The initial design of IBP and the subsequent development of the principles of logistical networking was initiated through joint work by Dr. James Plank and Dr. Micah Beck in the process of investigating technologies for the next generation of networking infrastructure.

While IBP can be thought of as a protocol for the management of appli-

cation level communication buffers, it can also be viewed as infrastructure for the remote access to shared files. This view allows for the development of application-specific file access strategies (e.g. staging, caching, replication,..etc) built on top of a unified standard infrastructure offered through IBP. This usage is further explored in this dissertation as well.

1.1 The end-to-end state control

The TCP protocol [30] at the core of many of the different protocols that drive the global Internet is built on the concept of *end-to-end state control*. This notion, as outlined in [33], is fundamental to the survivability of an established communication stream in the face of transient failures in the interconnecting network. The end-to-end principle maintains that any state required for the establishment and maintenance of a communication stream between two hosts connected to the Internet should be maintained at the two aforementioned hosts, and not on the many routers and switches involved in the actual packet transmission across the network. Specific examples of such state include the number of packets transmitted, the number of packets acknowledged, or the number of outstanding control permissions. This principle guarantees that if one or more failures occur in the inner fabric of

the network involved in the transmission of a communication stream, this stream can be reestablished using unaffected parts of the network without the need to restart the transmission. This follows from the fact that all information required to continue the transmission is maintained at the end hosts. Only the failure of one or both end hosts can lead to loss of this state, and subsequent failure of the communication operation. In addition, no state information that characterizes the movement of packets across the network is readily available to the end hosts. This also stems from the local nature of algorithms used in packet routing.

Th end-to-end principle proved crucial for both the fault resilience of the Internet as well as its scalability. Should such state be maintained on intermediate switches and routers instead, it would have been highly unlikely to devise scalable distributed algorithms that maintain such state in a coherent and accessible manner.

However, the gain in fault resilience and scalability achieved through the adoption of end-to-end state management has several drawbacks. As applications do not have access to *any* state and data maintained on intermediate switches and routers, it is not possible to influence where and when data may be stored (except at the end points). While this ability may not be required, or even feasible, at the packet level, the incorporation of such abil-

ity at a higher granularity allows applications to optimize data movement operations according to applications' needs.

Another drawback of the end-to-end approach stems from the reliable delivery nature of the TCP protocol. Should a packet get dropped or corrupted during transmission, it has to be resent from the communication source. This means that the same packet may have to traverse one or more upstream communication link more than once if the failure is caused by a link or congested router that is topologically closer to the receiving host. This can lead to resource utilization problems, especially for large files transmitted over large network distances, with a congested router or slow link connecting the receiver to the network. This problem can be greatly alleviated by progressively moving *the entire file* closer to the receiving host using storage depots that are part of the network fabric itself. This is indeed one of the possible uses of the IBP infrastructure proposed in this dissertation.

1.2 Web caching and the end-to-end model

As outlined in section 1.1, the end-to-end resource control model allows for network fault tolerance and scalability while limiting application flexibility and possibly causing resource waste. One of the areas in which these lim-

itations becomes apparent is access to objects on the world wide web. In the early days of the web, with relatively small number of sites and users, the need for direct control over the way in which a web document is delivered to the client was not recognized. However, as the number of users increased exponentially, it was recognized that by having control on how a document is delivered to the client, web site operators can achieve several favorable goals. Such goals include better bandwidth management and congestion control, improved user experience through better response time (i.e. faster document delivery), and cost reduction by avoiding the need to upgrade network connectivity infrastructure.

This realization has led to many projects that aim at altering the direct end-to-end model of document delivery. Projects such as Squid [79] and research work such as the projects at [1] and commercial entities such as Akamai [2] aim at addressing this issue by introducing *web caches* between the web server and client. In its simplest form, a web server tries to place its content at a close network topological distance to its clients to optimize the document delivery process. While this approach is clearly not in accordance with the end-to-end resource control methodology, the fact that it is done at the *file* level has allowed such approaches to materialize and succeed. However all such efforts remain at the application level, and are

implemented using application-specific mechanisms that are rarely interoperable. This dissertation suggests the use of IBP, as part of the network fabric, as a means to provide a uniform interface to state management that is better integrated with the network.

1.3 Shared storage access on the Internet

While state management in the network is considered the primary motivation behind the development of the IBP protocol, another important aspect of the protocol is that it enables efficient access to shared remote storage resources in a manner not available through any of the existing techniques. In this section, we will discuss the available methodologies for sharing distributed storage resources and outline some of the characteristics that render such techniques unsuitable for certain types of applications. This section motivates some of the design aspects of the IBP protocol which will be fully discussed in chapter 2.

1.3.1 Distributed file systems

The development of the Network File System (NFS) [70], the Andrew File System (AFS) [53] and its successor the Coda file system [71] in the mid-

eighties signaled the beginning of the wide adoption of the distributed file system concept. This was made possible due in part to the rapid developments in computing power and networking technology and protocols that took place starting in the mid-seventies. The efficient support of file access semantics (usually UNIX semantics [51]) in a distributed setting has been the major design objective of various distributed file systems. This requirement generated a set of new problems for file system designers (e.g. distributed file locking) and added to the complexity of existing ones (e.g. protection and security, directory management). The need to achieve acceptable performance in a distributed file system usually translates into the need for elaborate caching policies that contribute to the complexity of the aforementioned problems.

While distributed file systems provide a convenient mechanism and a familiar interface for access to distributed storage under a single administrative authority, problems arise when attempts are made to extend this model to independent administrative domains. System administrators will not usually agree to mount file systems on hosts they do not control for security reasons.

Distributed file systems provide a fairly high level access to distributed storage. The physical location of a user's home area is statically determined.

As a result, a non-privileged user (and/or application) cannot exploit physical aspects of the system (e.g. proximity, server load, ..etc.) to store data in such a way as to improve application I/O performance.

With the proliferation of the Internet and the ubiquity of the World Wide Web, another class of distributed file systems has emerged. The "*wide area*" distributed file systems aim at providing familiar file I/O semantics to files that are already accessible through an Internet protocol (e.g. HTTP, or FTP). It should be noted that nothing in the specification of traditional distributed file systems prevents them from wide area implementation. Protocol overhead and characteristics make such an implementation in a low bandwidth, high latency network impractical. Projects such as Jade [64], WebFS [78], WebNFS [25], and UFO [6] build a distributed file system on top of existing Internet protocols. However, these systems remain constrained to publicly accessible content and are further constrained by limitations to the underlying transport protocols. In addition to the aforementioned problems (which are made more difficult by the heterogeneity of the underlying protocols), the problem of maintaining a global name space becomes especially important.

From the above, it can be seen that while distributed file systems provide a convenient mechanism for shared access to distributed storage, such

convenience is limited to users within a single administrative domain. A combination of administrative and performance issues limit the usefulness of file systems built on top of existing Internet protocols. Distributed file systems in their current form do not offer a viable solution through which storage resources can be shared among users who belong to different administrative domains.

1.3.2 Internet shared storage

The Internet currently offers wide support for a set of protocols that allow for the shared access of storage among various users. Protocols such as FTP [61], HTTP [38], and NNTP [47] have been developed to support a *publishing* mode of shared storage access. An application *publishes* content to a server (or multiple servers in the case of NNTP), where it can be retrieved through the relevant protocols. While this model has worked well (and continues to do so), it has several deficiencies that render it unsuitable for more generalized use:

- **Administrative overhead:** For general public access to storage via FTP, an *anonymous* account needs to be in place. Setting such an account may not be in line with many organizations' policies, which

may favor strict control over resources. In the case of web servers, the anonymous mode of access is the default one. However, administrative consent may be required for placing content in the public domain via HTTP. In addition, public access to such storage is often restricted to read-only access, with write privileges closely monitored by the organization's web managers.

- **Granularity:** Access to stored objects via FTP, HTTP, or NNTP is done at the object level. No mechanisms are present in the protocols to allow access at a lower granularity. This limits the flexibility afforded to applications which may be interested on segments of the stored object. This aspect leads to resource over-utilization as entire files are transferred between network end points, even though the receiving (e.g reading) application requires only a *known* portion of the file.

- **Access control:** Limited or no access control is available to users and/or applications that store content using any of the aforementioned protocols. Anonymous FTP access, by definition, allows unrestricted access to the stored objects by any FTP client. Access to objects through HTTP can be restricted through the use of user authentication mechanisms that are part of the protocol. However, such control

requires administrative intervention and is awkward to apply at the file level. Access to content stored on NNTP servers is available to any news client.

From the above discussion, we can conclude that while existing protocols allow for various publishing modes on network-attached storage, they offer little utility for true public sharing of storage resources between users and owners of such resources. For such sharing to be possible, a different storage model needs to be developed that addresses the points mentioned earlier, while maintaining certain level of owner control over local storage resources. This led to the development of the **I**nternet **B**ackplane **P**rotocol (**IBP**), which attempts to provide an alternative storage model suitable for shared public access.

1.4 Related work

This dissertation deals with the development of a shared storage access protocol and infrastructure as part of the network fabric. As such, this research touches upon several areas in distributed systems and networking. The augmentation of traditional networking with user-specific computations has been the focus of research into *Active Networks* [77]. In an active net-

work, switches and routers preform customized computations on the messages flowing through them. The networks become active in the sense that nodes perform computations on, and modify, the packet contents. In addition, this processing can be customized on a per user or per application basis. This migration of computations from the edges of the network toward its interior is in contrast to the traditional packet networks, where user data is passed without inspection or modification. In [49], Legedza et al present an active networking protocol that uses caching within the network backbone to reduce load on servers and backbone routers. The relation between active networking and the end-to-end argument is discussed in [20], where Bhattacharjee et al argue that active networking is a natural consequence of the end-to-end argument, because certain functions can be most effectively implemented using information that is only available inside the network.

We submit that the IBP protocol introduces the element of mass storage into the network-activation paradigm. The addition of standard remotely accessible shared storage depots as part of the network fabric allows traditional (i.e. edge) applications as well as active networks type codes to expand the space of operations that can be efficiently performed *in the network*.

The *application level routing* of end-to-end data over wide area network

has been receiving increased interest in the network and distributed systems research communities. In [31], Chae et al argue that as networks become larger and more heterogeneous, situations arise in which the ability to identify particular topological properties enables capabilities and performance that are difficult to achieve with a purely ”black box” interface to network topology. They propose an approach to query and synthesize network information that allow constrained programmability.

The integration of application-level routing and storage nodes is explored by Rowstron and Druschel in [68] for the design of Pastry. The Pastry system performs application-level routing and object location in a potentially very large overlay network of nodes connected via the Internet. The system is designed to be used to support a variety of peer-to-peer applications, including global data storage, data sharing, group communication and naming. Pastry is used as the foundation for PAST [66, 67], a large-scale, persistent peer-to-peer storage utility.

Another project that combines distributed storage and application-level routing is the Tapestry project [84]. It is an overlay location and routing infrastructure that provides location-independent routing of messages directly to the closest copy of an object or service using only point-to-point links and without centralized resources. This application-level routing layer is

the foundation of the Oceanstore system [48], a utility infrastructure designed to provide wide area continuous access to persistent information using untrusted servers. The system uses data redundancy and cryptographic techniques for data protection, and data caching to improve performance.

The computational grid concept [40] with universal transparent access to distributed computing resources at its core is another area of research where combined efficient utilization of computing, networking, and storage resources lies at the core. In that regard, many projects that focus on efficient distributed storage access in support of wide area scientific computing have been carried out. As part of the GLOBUS project [44], Bester et al [19] discuss the design of a service to enable Global Access to Secondary Storage, or GASS. The GASS system defines a global name space via Uniform Resource Locators and allows applications to access remote ”files” via standard I/O interfaces. The GASS system aims at providing optimized support for file access patterns common in grid computations and the ability for user controlled management of bandwidth. The Data Grid [7], is a more recent effort that aims at the development of large scale data management architecture that builds a virtual global storage space on top of local storage systems that are made available to the computational grid. The project provides abstractions for storage systems that are combined through a metadata

layer to enable the provisions of various data grid services such as replica management.

The aforementioned projects and many others in the distributed storage management are built on special purpose protocols and infrastructure that do not allow for easy inter-operability. We submit that the IBP distributed storage infrastructure provides a middle layer that allows these projects and others to access remote storage resources in a uniform, flexible way. The IBP protocol provides the foundation on top of which many remote storage and network management policies can be implemented.

1.5 Dissertation organization

This dissertation focuses on the development and utilization of the Internet Backplane Protocol for access to remote storage resources. In chapter 2, we present the details of the IBP protocol and discuss how the main characteristics of the protocol can be used to enable higher level policies and applications. In chapter 3, we discuss the use of the IBP protocol to optimize the delivery of email messages that contain large attachments. This demonstrates the utility of combining network topology information and IBP enabled storage resources to develop data movement policies that bal-

ances execution time and resource utilization. In chapter 4, we present the use of IBP storage as enabling infrastructure in support of wide area scientific computing. This use adds the third dimension of computing time to the elements of networking and storage resources to optimize the overall throughput of a wide area task farming application. In chapter 5 we provide conclusions and suggestions for future work.

Chapter 2

The IBP protocol and infrastructure

This chapter presents the details of the IBP protocol and infrastructure. The first part of the chapter discusses the distributed storage paradigms made possible by the availability of the IBP distributed storage access protocol. The second part of this chapter discusses the details of the IBP infrastructure and protocol. The final part presents an outline of applications made possible through the use of IBP. The work presented in this chapter has been published in [59] and refined in [58].

2.1 Introduction

The *Internet Backplane Protocol* is designed to allow user-level applications the capability to control the manner in which data is stored, moved,

and accessed across wide area networks. This control has traditionally been encapsulated in an *end-to-end* resource control model that is not easily accessible from the application layer in the communication protocol stack. The standard TCP/IP protocol stack is an example of such model, where applications have little or no control over the route or intermediate buffers used in the movement of data between two network endpoints.

The *Internet Backplane Protocol* is designed to allow user-level applications to manage distributed data storage and movement as a network service that is closely linked to communication services offered by the network fabric. 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. As outlined in chapter 1, the scalability problem is addressed by exploiting this functionality at the *coarse grain* application level data entities, i.e. files and file segments, rather than at the *fine grain* level of packets used in the standard TCP/IP communication protocols.

In the design of the Internet, considerations of robustness and scalability have led designers to opt for a *stateless* model. This model is the equivalent of the *functional model* of computation. In a functional computational model the internal state of each system element is hidden behind a functional

interface. In such a model, the failure of one element results only in the failure of the current transaction and the (possible) future unavailability of the failed element. By contrast a *shared-memory model* of computation is one in which long lived dependencies exist between elements. Such dependencies can compromise both robustness and scalability of a large distributed system. This model corresponds roughly to a *stateful* networking model, in which a failure that corrupts the shared system state could affect future transactions, and the size of the system is limited by the need to coordinate the sharing of system state correctly. However, the shared-memory model does allow applications to have direct access to various system elements, thus facilitating more application and domain-specific optimizations.

While the design of the Internet adopts the stateless (or functional) model, the designers of many modern large scale information systems have been incorporating elements of the stateful (or shared-memory) models into their systems. This can be seen even in systems built on top of the standard stateless Internet infrastructure. The move toward a more stateful design is driven primarily by the need for better control and optimization of data movement and access patterns. These features are difficult to express in functional terms. This move can be clearly seen in the proliferation of web content caching and replication mechanisms (e.g. [23]). Another area

where this trend can be observed is the distribution of massive scientific datasets [12, 24, 41] via networked large storage resources.

However, shared state management at the application level in networked systems has been usually done using application specific mechanisms. Those mechanisms are rarely inter-operable, which lead to the balkanization of state management capabilities and made standardized interoperability among applications difficult to achieve. The Internet Backplane Protocol is designed to provide a standard mechanism upon which application state management capabilities can be built.

The Internet Backplane Protocol can be seen as a mechanism to manage either *communication buffers* or *remote files*. Both characterizations are equally valid and useful in different situations. In general, we can view IBP as a mechanism for the management of elementary storage entities, called *byte arrays*. The two views of byte arrays as communication buffers or network files are presented below:

- **IBP as buffer management.** In packet-switched networks (of which the Internet is a prime example) information flows between nodes in the form of *packets*. Those packets are moved incrementally closer to the target node by means of the routing algorithms implemented

(usually in hardware) in network routers. Those routers adopt a *store and forward* mode of operations, in which incoming packets are stored in router buffers (usually in FIFO queues) pending a decision by the routing algorithm as to how any particular packet is to be forwarded en route to its destination node. Hence, we can view those router buffers as a form of storage in the network that is only accessible by the routing algorithms themselves.

Similarly, IBP byte arrays can be viewed as application-managed communication buffers in the network. The IBP architecture accommodates time-limited storage and FIFO queues as means to optimize utilization of finite storage resources. Applications utilize IBP byte array buffers to actualize data staging and routing at a higher granularity than that used at the packet level through network routers.

- **IBP as file management.** The storage of large files is usually accomplished through the use of highly structured file systems that are typically private to one or more network-connected hosts. Access to such files by applications that do not reside on the set of *owner* hosts is usually accomplished through non-standard techniques that do not allow for easy inter-operability.

> In this regard, IBP byte arrays can be viewed as building blocks of file(s) that reside *in the network*. As such, IBP allows an application to have read and/or write access to remote files, as well as the capability to direct the movement of such files remotely. The storage space accessible through IBP can be thought of as a shared storage space in a manner analogous to the sharing of bandwidth among different file transfer operation.

IBP, in allowing data to be stored at one location while en route between two network nodes, introduces the ability to control data movement *temporally* as well as *spatially*. The spatial aspect of data movement control is exercised by packet routing algorithms as they strive to transfer an incoming packet to another router (or to its final destination) as soon as possible. Thus, the routing of packets through the network is made through a sequence of *spatial* choices, as an incoming packet is routed on one of several alternative links.

IBP, on the other hand, allows for data to be stored at intermediate location(s) while en route from sender to receiver, adding the ability to control data movement *temporally* as well as spatially. This ability to control data movement both spatially and temporally is termed *logistical networking*, in

analogy with the system of warehouses and distribution channels commonly used in the logistics of transporting industrial materials. 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. In the following section, the details of the IBP architecture that make such paradigm feasible are presented. We describe the API that allows client applications to access IBP functionality as well as different configuration options for the IBP storage depots that makes it amenable to different modes of utilization.

2.2 IBP structure and client API

IBP has been designed as a client-server infrastructure. An IBP server is designed to represent a *storage depot* that can be used as an intermediate storage hub in a logistic networking application. We have designed the IBP server in such a way that it does not require any special privileges or superuser credentials. As such, any ordinary user can set up one or more storage depots using his/her personal accounts. In addition, he/she can have access to IBP storage depots owned and exported by colleagues/collaborators. This notion of shared access to storage is at the core of the IBP infrastructure and

will be outlined in more detail in subsequent sections. Another prominent feature of the IBP infrastructure is the minimalist approach to storage management and access. We include within the IBP system a set of primitive operations that can be used as the foundation for more elaborate storage and logistical networking schemes. The fundamental IBP operations are:

1. Allocating a byte array for storing data.
2. Moving data within the space of IBP servers, or between the IBP server(s) and outside clients.
3. Management of a previously allocated IBP storage resource.

The client API for IBP consists of seven procedure calls for storage allocation, data movement, and allocated storage management. The IBP client forwards user requests on to one (or more) IBP server (interchangeably called *IBP depot* for its role in logistical networking). The server daemon is designed to be highly configurable, allowing individual users to have control over the manner in which local storage is exported for use in IBP applications. In the initial IBP prototype ,connections between clients and servers are made through TCP/IP sockets. However, the IBP infrastructure can be implemented on top of other transport protocols (e.g. UDP) to bring its

functionality closer to the network. IBP servers do not require client authentication as part of the IBP protocol. The storage managed by an IBP server can be accessed by any client that can locate, and attach to, the server.

2.2.1 IBP storage characteristics

An IBP server allows clients access to local storage resources. Clients are presented with a flat name space, with no client-assigned file names or directory structures. Such constructs are best implemented in a layer that is built on top of IBP. The characteristics of storage available through IBP can be considered as three dimensional space, with one axis representing *durability*, another axis representing *reliability*, with the third axis representing *storage layout* as seen in Fig. 2.1

The durability of a storage entity on an IBP server can have one of two characterizations. A storage entity can be *permanent*, which indicates that it is kept on the IBP server until it is explicitly removed. *Time-limited* storage on the other hand is re-claimed by the IBP storage server upon expiration of its declared lifetime. As for reliability, a *stable* storage entity can *only* be removed upon explicit request from an IBP client with proper capabilities. Alternatively, *volatile* storage can be re-claimed by the IBP server at any time (with no explicit client command to do so). The third dimension of

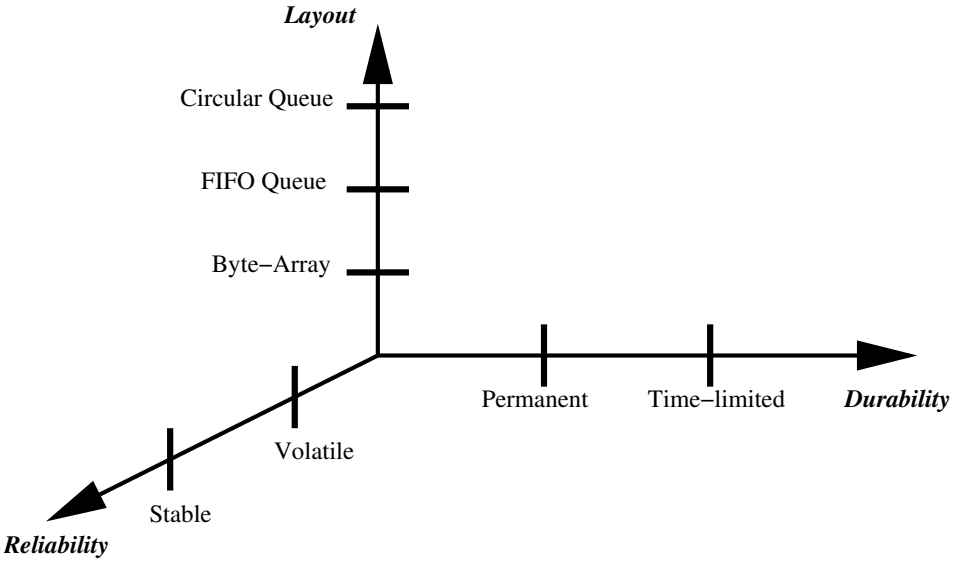


Figure 2.1: IBP storage characteristics

storage characteristics define the different layouts and access semantics of IBP storage entities. A *byte-array* is an append-only storage entity that supports random access reading. A maximum of one append operation can have access to an IBP byte-array at any given time. There exists no limit on the number of read requests that can be simultaneously active accessing a single IBP byte-array. An IBP *FIFO queue* is analogous to the UNIX `fifo` special file, with a maximum of one single reader and one single writer active at any given time. Append requests to an IBP FIFO queue proceed while available storage in the FIFO queue exists to satisfy them. Further append requests (or the remainder of a partially fulfilled request) are blocked pend-

ing the availability of "empty" storage space (through the execution of read request(s)). Read requests to an IBP FIFO queue proceed as long as data is available to satisfy them. Attempts to read from an "empty" IBP FIFO queue are blocked pending the availability of new data to read (through another append request). An IBP *circular queue* is a fixed-size FIFO queue, with write requests that exhaust available empty storage proceeding to overwrite the earliest written data in a circular pattern.

IBP Clients initially gain access to IBP storage entities 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. In the current implementation of IBP, each capability is represented as 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 2.2. We omit error handling for clarity. The full API is described in a separate document [36].

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 2.2: The IBP Client API

Allocation

The main goal of IBP is to provide a shared storage space that is embedded into the fabric of the network. Elements in this storage space (i.e. IBP servers) are owned and controlled by individual users who provide access to storage that they "own" in exchange for getting access to other users' storage resources. This enables users, both from within and outside any one administrative entity, to have access to network storage in excess of that which they would normally have access to using traditional techniques

(i.e. by having remote accounts). This shared storage models necessitates the deployment of a resource allocation and de-allocation mechanism that maintains a balance between user control over his/her own resources, and the sharing of such resources with the outside world. This allocation scheme must necessarily be different from typical allocation policies implemented to manage storage attached to one or more host systems under the control of a single administrative authority. We view the use of IBP to facilitate the sharing of storage resources as analogous to the sharing of bandwidth among various network connections. It is thus helpful to consider some of the issues involved in resource sharing in the Internet, and how they compare with the allocation of 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. Policies and procedures are developed and deployed to guarantee availability of such resources to their "owners", or a proper pricing scheme is implemented to compensate them for others' utilization. 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 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 a storage allocation scheme that incorporates some of these administrative limits, while at the same time seeking to provide guarantees that are as strong as possible for the client.

The administrative limits on storage resource allocation are represented by the storage characteristics outlined in section 2.2.1. The owner of storage space exported through an IBP server determines the "proper usage" policy of his/her resources by specifying the total storage space available for allocation in each of the allocation categories outlined earlier. The details of this aspect of IBP server configuration are presented in section 2.2.2. The different categories of storage durability and reliability allows IBP client to devise storage management schemes that best meets application needs. 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()` shown in figure 2.2. The `host` parameter defines the IBP server on which the allocation is attempted. The IBP client needs to have an out-of-band mechanism to locate IBP servers in the network. The IBP infrastructure does not include a naming and location service, however such service can be built on top of IBP. Indeed one such service is used in the design of IBP-enabled mail service discussed in chapter 3. The `size` parameter is used to specify the maximum amount of storage that can be used by the newly allocated IBP entity. The attributes of the allocated IBP storage are specified in the `attr` parameter. Those attributes include the storage type, reliability and durability outlined earlier in section 2.2.1.

Successful completion of the allocation process indicates that under the storage management policy implemented by the IBP server running on the aforementioned host, there exists enough storage that meets the requirements of the client. Upon successful allocation of storage on the IBP server, a trio of capabilities is returned to the allocating client. These capabilities allow the client to perform read, write, and manage operations on the newly allocated IBP storage.

Reading / Writing

All reading and writing to IBP byte arrays is done through the four reading/writing calls in Figure 2.2. The calls are designed to facilitate data flow between IBP clients and IBP servers (`IBP_store()` and (`IBP_read()`), between IBP servers (`IBP_copy()`), and between IBP servers and *third party* data sources and sinks (`IBP_remote_store()` and `IBP_deliver()`).

All IBP calls require the client to present the appropriate capability the the IBP server(s) involved. The calls `IBP_read()` and `IBP_store()` allow IBP clients to transfer data directly between their main memory storage and IBP buffers accessed through the appropriate capabilities. `IBP_store()`, as well as all other IBP write calls, have append semantics. The `offset` parameter in IBP read calls allows such operations to access any location in an IBP byte array. This parameter is ignored in calls that read from IBP FIFOs or circular queues, as read location is an architectural property of these IBP storage entities.

The IBP infrastructure incorporates operations that allow client-directed data transfer to IBP servers from external data *sources*, as well as data delivery from IBP servers to external data *sinks*. These operations are among the features that distinguish IBP from other network-based storage schemes

and distributed file systems. In both of these operations, a target *port* on the third party host is required as part of the client API. This allows for increased flexibility in delivering and/or receiving IBP data from a wide variety of user and system applications. IBP data is delivered or received in raw format using TCP/IP sockets, with no other protocols to implement on third part applications.

The inter-IBP data movement is controlled through the `IBP_copy()` call. This call reads data accessed through the IBP source *read-enabled* capability and writes it to the IBP storage accessed through the target *write-enabled* capability. The two capabilities do not have to access the same type of IBP storage. The append semantics of IBP write operations alleviates the need to specify a target write address. The `offset` parameter for the read-side of the copy operation is ignored if the source capability does not access a *byte-array* type IBP storage. It should be noted here that an IBP management capability carries implicit read and write permissions, allowing such a capability to be used for both read and write access to IBP storage.

In all IBP operations, error conditions are flagged by a negative return value on the client side. A diagnostic error code is stored in the special variable `IBP_errno`. Upon failure, this variable can be checked for diagnosis, and an appropriate error text message can be retrieved via the

`IBP_strerror()` call.

Management / Monitoring

The management of allocated IBP storage is achieved through the use of the `IBP_manage()` client call. This call requires the client to pass the IBP management capability associated with the target IBP storage area. This capability is part of the trio of capabilities returned to the IBP client upon successful completion of the `IBP_allocate()` client call. `IBP_manage()` allows the client to decrement or increment the reference count associated with the read and write capabilities of an IBP storage area. If the reference count associated with the write capability reaches zero, the IBP storage area becomes read-only. In addition, if the reference count associated with the read capability reaches zero - indicating no client can read data stored into the IBP storage area - the storage area is reclaimed by the IBP server and is no longer available for future client access. This client-directed storage area removal is used in conjunction with the automatic server-side storage area deletion triggered by the lifetime expiration of time-limited storage, or server reclaiming of volatile storage. It should be noted that client-held read or write capabilities associated with deleted IBP storage areas are invalidated and an error condition arises if such capabilities are used to access

their associated IBP storage.

Another utility of the `IBP_manage()` call is to probe and modify, the state of existing IBP storage areas. These properties include maximum usable storage available to the IBP storage entity and the lifetime of time-limited storage. Reducing the maximum storage size below the size of data already stored in the IBP storage area results in irrecoverable data loss. The reliability or the storage type characteristics of an IBP storage area cannot be changed via the `IBP_manage()` call.

2.2.2 IBP server configuration

An IBP server allows IBP clients access to server-side storage through the IBP protocol. The IBP server does not require special privileges or super-user status. The server implements a storage allocation policy set by the storage owner. This policy is specified through a server configuration file that specifies the maximum available storage available for both volatile and stable storage, the local directories used to store IBP entities, and the maximum duration any IBP storage entity can exist on the IBP server.

The IBP server also implements a failure-recovery mechanism that allows access to IBP storage to be restored, even after an IBP server crash or failure. The information needed to reconstruct server state is stored in the

header of every IBP storage entity stored in a permanent storage directory.

2.3 Application strategies enabled via IBP

As outlined earlier, IBP facilitates a combined view of networking and storage resources that opens the door for new applications and improves the efficiency of some existing ones. This combined network-storage view, which we term *logistical networking* [17] is centered upon the location of IBP storage depots at strategic, application-specific, network location. Application developers devise and implement data staging and movement policies, using IBP depots, to enhance overall application performance and/or functionality. IBP's capability-based storage access allows applications explicit control over overall resource allocation and utilization. In this section, we present some logistical networking strategies that demonstrate the utility of IBP in several application domains. Two strategies are further expanded in the following chapters.

2.3.1 Sender-side data storage

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.

This strategy alleviates the need for sender-side applications to block for extended periods of time pending transmission of their state to remote sites. Data is written to an IBP depot connected to the sender via a high bandwidth, low latency connection. Such an IBP depot is typically co-located with the sender(s). This configuration guarantees that senders block for as little time as possible while exporting their state to the receiver side (or consumer). Remote receiver applications retrieve data from the server-side IBP depots independent of the writing process. This effective decoupling of sender and receiver applications allows the two sides to proceed at different rates of execution. This scheme can further be used to reduce the amount of data transferred over the wide area network, as the receiver app can make decisions as to which data items (if any) are needed from all data item stored by the sender application.

The utility of such paradigm can be seen in two representative applications, remote gathering of network sensors information and checkpointing of long running scientific applications. The Network Weather Service (NWS) is a system that monitors network resources and predicts their future behavior [81]. 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.

NWS, as well as similar remote measurement systems, can use a sensor-side IBP storage depot to store sensory measurement. IBP circular queue storage can be used to overwrite aging measurements with more recent ones. Remote monitoring and forecasting applications have a great degree of flexibility in managing sensors' data stored on IBP servers. The use of the IBP protocol for the management of the sensors' measurement allows data replication and consistency policies to be implemented on top of a standard infrastructure layer, which reduces the overhead involved in the development and deployment of such policies .

The collection of distributed sensor data is one instance of a more gen-

eral problem: the collection and maintenance of continuously produced data from various distributed sources. A similar type of application that benefits 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 [4, 35, 56]. However, for reasons of performance, migratability, availability and added fault-tolerance, it is often useful to employ a more flexible storage medium [63]. IBP is a natural facility for managing this storage, providing most of the necessary primitives for this task. Checkpointing to IBP depots that are *close* to the long running program(s) reduces the overhead involved in performing each checkpointing operation. The ability to transfer checkpoint data through a high bandwidth, low latency connection to a *local* IBP server reduces the time it takes to commit any single checkpoint, thus improving overall system fault tolerance. IBP-enabled checkpointing has been added to the core of the NetSolve server software library [5], allowing for more flexible management of checkpoint data. Using IBP, the location of the checkpoints can be managed by the Netsolve client or agent (see section 2.3.3) in a manner that is independent of the checkpointing mechanism and server software. This in turn facilitates

process migration and scheduling in addition to fault-tolerance.

2.3.2 Receiver-side data storage

A content delivery system built on top of a collection of IBP depots dispersed across the network can operate by progressively moving content *closer* to the receiver. The sender delivers its local content to a local IBP sever. The content delivery system is then responsible for making such content available at any given time to the receiver. This is accomplished through a client pull mechanism that uses the IBP server currently holding the desired content as a proxy content server. Simultaneously, the content is moved progressively toward the receiver(s) using the collection of available IBP servers as intermediate hubs. In essence, such a system performs user-level routing at a very large granularity (that of the entire object being transmitted).

Such a content delivery mechanism can be applied to different modes of object transmission across the Internet. In chapter 3 we present the details of such a system applied to facilitate the delivery of large email attachments. A similar mechanism can also be applied to other content delivery systems through *speculative transfers*.

Speculative transfers can be used to improve throughput in any informa-

tion 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. Client-based (or pull) techniques have 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 server *push* implementation of the same strategy allows the server to retain control of the process, transferring speculatively only when it is otherwise idle. It also allows the server to transparently include objects stored in geographically dispersed depots into a given transfer. In addition, if speculative transfers target local IBP depots rather than the client's storage, there is no impact on the client's own resources.

2.3.3 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.

NetSolve [29] 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 depicted in Figure 2.3. 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.

The functional nature of the traditional NetSolve computational model incorporates no state management in the server. NetSolve servers receive arguments to calls from clients. Arguments are processed and the result is conveyed back to the initiating client. Incorporating state management into

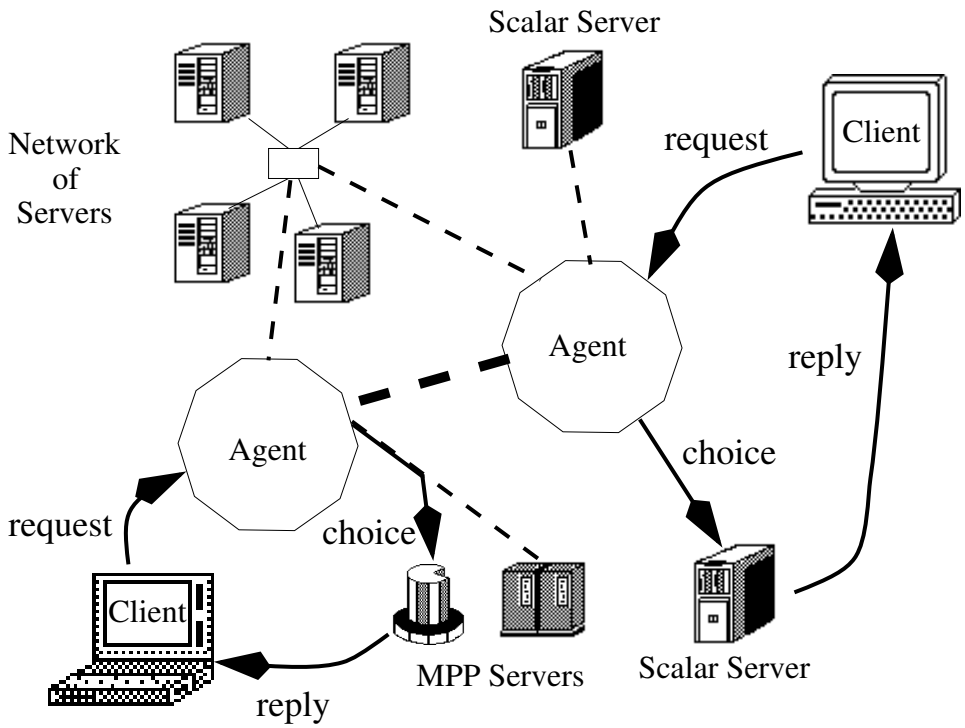


Figure 2.3: Basic structure of NetSolve

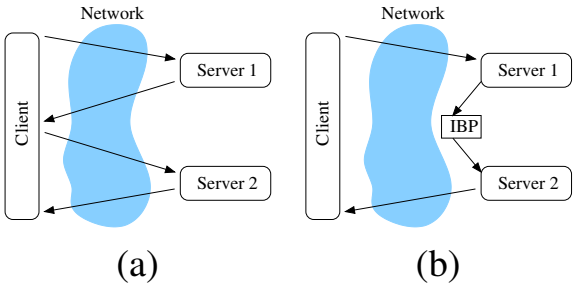


Figure 2.4: Scheduling consecutive NetSolve calls with a common argument. (a) No state management, (b) Storing temporary results in IBP near the computational servers.

NetSolve allows more complex interaction scenarios, where intermediate results are stored in IBP depots. This in turns allows for co-scheduling of storage and computations on one or more NetSolve servers. 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 2.4(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 2.4(b)). Using IBP depots for the storage of intermediate results maintains clients' ability to access and inspect those results, through appropriate IBP capabilities exposed via the NetSolve servers. The incorporation of intermediate state

management into the NetSolve model is explored in [8].

Chapter 3

IBP-enabled wide area content-delivery via email attachments

3.1 Introduction

Wide area content delivery has traditionally relied on the establishment of a direct client-server network connection. Using this connection, clients can request delivery of certain content (*pull*), or the server can transmit predetermined content to the client (*push*). Electronic mail has been one of the earliest, and most commonly used, push-based content delivery mechanisms. The sender (which is considered the content server) transmits a message (the content) to one or more recipients (content clients). This is typically done using the SMTP protocol [61]. In a typical email transac-

tion, the sender uses a Mail Transport Agent (or MTA) to connect to a local SMTP daemon (commonly called a mail gateway). The sender-side SMTP daemon stores the outgoing message in a local spooling area, while making connections to the (one or more) recipient's SMTP daemon to deliver the message. The message is removed from the sender-side SMTP daemon only after delivery confirmation is received from all receivers' SMTP daemons. Messages are stored in the spool area of receiver-side SMTP daemons until they are by the receiver's MTA, and deleted by the receiver.

The standard email content delivery relies on the concept of *spooling*, or storing of outgoing and incoming messages in storage attached to SMTP daemons. These spool areas are shared by all users who use these SMTP daemons as their mail gateways. This sharing of storage resources has led to the development and implementation of policies that limit the size of any single email message accepted by the SMTP daemon. In this chapter, we present a content delivery mechanism for large email attachments that utilizes the logistical networking infrastructure provided by IBP depots. We discuss the different alternatives that exist for the wide area delivery of large objects and outline the design of an IBP-based system that utilizes logistical networking concepts to ameliorate the resource sharing problem of standard email delivery mechanisms. The IBP-based solution utilizes user-level rout-

ing techniques to move large email attachments closer to their recipient(s). This approach maintains the push-based delivery of the textual portion of an email message, while introducing pull-based retrieval of the (usually large) non textual attachment. Moving the attachments closer to the recipients reduces the pull time experienced by the recipient, provided that the pull operation is initiated after the routing algorithm has finished executing.

3.2 Traditional point-to-point content delivery

The delivery of content in a point-to-point communication has been traditionally accomplished using electronic mail. This mode of content delivery is different from the *publish* mode, where content is made available on a server for general or restricted access. This chapter focuses on the point-to-point content delivery, while recognizing that publish-based content delivery mechanisms can be tailored for point-to-point delivery. In this section, we discuss the alternative mechanisms that can be employed to achieve point-to-point delivery of large digital objects and discuss the limitations of each approach.

3.2.1 Email with attachments

The most commonly used and convenient point-to-point content delivery mechanisms involve the use of Email attachments. Since the standard Simple Mail Transfer Protocol (SMTP) [46] mandates the use of textual email message payloads, tools have been developed to encode non-textual objects into ASCII text suitable for transport via SMTP. One such early tool is Unix's `uuencode`. `Uuencode` is used to encode any binary data file into an ASCII message suitable for transport via SMTP. `Uudecode` is also used to decode the ASCII text back to its original binary format at the receiver end.

The MIME protocol [22,42,65] was later developed as a standard format for attaching non-textual files to mail messages and identifying their types so that mail clients can bundle multiple data files into an email message, send them to recipients, and then have the recipients unbundle them and launch file-specific applications to act on them. This development standardized operations that had been earlier carried out in an ad-hoc fashion using `uuencode` and similar tools.

The development of the MIME standard has simplified the processing of non-textual attachments at both the sending and receiving ends. However, it

has not addressed the fundamental resource utilization problem manifested in the spooling nature of SMTP daemons. The ASCII encoding of binary attachments typically increases their (already large) sizes by a factor of the order of 1.4. Spooling storage available to the SMTP daemons at both the sender and receiving ends is finite, and is shared by all users. Another factor that contributes to the resource utilization problem is the lack of standard mechanism that allows multiple recipients to share access to the same copy of a binary attachments. Standard SMTP daemons store a separate copy of the email message and attachment(s) for each recipient, even if they all share the same SMTP daemon. These factors have led system administrators to impose limits on the total size of email messages accepted by SMTP daemons. As such, the use of standard MIME attachments for the point-to-point delivery of large binary content is untenable for objects that exceed the size threshold imposed by the SMTP daemon system administrators.

3.2.2 Sender side data staging

In this mode of delivery, the sender stages the binary content on a content server accessible from the ”outside world”. The textual message is then used to transmit a pointer, as well as any required access information, to the recipient(s). The recipient(s) would then retrieve the content in question

from the server and notify the receiver that content had been successfully retrieved. The sender would then remove the content from the server. This can be typically accomplished using HTTP [38] or anonymous FTP [62] servers.

This approach overcomes the limitations of the spooling technique adopted by standard SMTP daemons. Furthermore, it alleviates the resource utilization problem since only one copy of the binary content is stored and shared by all recipients. This solution, however, has limitations that limit its usefulness. The sender needs to have write access to a public content server. This access may not be readily available in all organizations. The need to restrict access to the binary content to its intended recipient(s) adds further burden on the sender (e.g. the need to setup password-protected web page, or non-anonymous FTP server accounts). Such measures may not be applicable due to administrative or usage policies. Another drawback is the lack of mobility of the binary content during the elapsed time between sending the textual SMTP message, and the recipient downloading the digital object. This downloading occurs from a server that is typically ”local” to the sender, causing a wide area transfer of a large digital object. Such wide area transfer can be potentially slow, as the transfer crosses many networking hubs increasing the potential for packet loss and re-transmit from the origin

server. In addition, there is no automatic mechanism to inform the sender when the binary file has been successfully downloaded, and can be safely removed from the content server.

3.2.3 Receiver side data staging

This technique is similar to the one discussed in section 3.2.2, however in this scenario, the public content server is managed by (and usually local to) the recipient. A typical example of this is when the receiver owns an anonymous FTP server with an `incoming` directory in which anonymous users may upload files. Alternatively, the receiver may have a special web page with file upload capability. The sender then uploads the file and mails the receiver a pointer to it. The receiver may then download the file upon reading the mail, and delete it when the download is finished. This solution overcomes some of the limitations of the sender-side staging technique. However it still requires a wide area transfer of the binary attachment *before* the textual message is sent, which may be a lengthy process for very large files. This approach shifts the administrative responsibility for the content server to the receiver, and it requires the receiver to open up such server for write (either anonymously or through a more controlled process). This introduces the same access and administrative overheads associated with the

sender-side staging solution.

3.2.4 Logistical networking solution using IBP-Mail

The availability of distributed writable IBP storage depots makes possible a solution the the point-to-point content delivery problem using a combination of traditional SMTP messages and user-level routing of binary attachments. In this approach, henceforth called IBP-Mail, the storage depots are registered with IBP-Mail agents. The sender registers with one of these agents and stores the file at a nearby depot. The sender then mails a message with the file's ID, as assigned by the agent. Meanwhile, the agent uses user-level routing techniques to attempt to move the file to a depot close to the receiver. When the receiver reads the message, he or she contacts the agent to discover the whereabouts of the file, and downloads it from the appropriate depot. The file may then be deleted from the depot.

The fundamental difference of the IBP-Mail solution from other options stem from the use of IBP depots as temporary content servers for the binary content while in transit. This allows for optimized content delivery to the receiver from a close IBP depot provided sufficient time elapses between the delivery of the textual SMTP message containing the binary file ID, and the receiver attempting retrieval of the IBP-Mail attachment. A worst case sce-

nario would have the receiver immediately attempting such retrieval upon receiving the SMTP message, and before the IBP-Mail system has had a chance to move the attachment closer to the receiver. This would be equivalent to the solution presented in section 3.2.2, but with the administrative overhead and lifetime management issues handled transparently by the IBP-Mail system. In the following section, we present the details of the IBP-Mail architecture and discuss the IBP features that make such solution a usable and preferred option.

3.3 The structure of IBP-Mail

The use of IBP depots and logistical networking to facilitate the delivery of large binary content in a point-to-point transfers allows *correctness* to be maintained at all times, while *performance* is optimized whenever possible. The recipient is always able to retrieve his/her intended attachment, preferably (but not necessarily) from a nearby IBP depot. For this technique to work, the client-side (sender and receiver) tools needed should be easily deployable and can be potentially integrated into standard email clients. In this section, we detail the structure of IBP-Mail that allows it to be used for the solution of the problem outlined in section 3.1.

IBP-Mail consists of three components, as depicted in Figure 3.1. These are:

- **A pool of IBP depots**. Ideally the pool will consist of a collection of servers with wide geographic (and network topological) distribution. It should be noted however that the IBP-Mail system is designed to operate correctly with a single IBP depot in the pool.
- **IBP-RQ: A registration/query server**. This is a server that maintains the state of the IBP server pool. IBP depots are registered with the IBP-RQ server, and clients query the IBP-RQ server for information about its registered pool of IBP depots. More information about the IBP-RQ server is given below.
- **IBP-NT: An agent for naming and transport**. The IBP-NT keeps track of where the data file is in the IBP server pool, and directs the movement of the file from server to server. More information about the IBP-NT agent is presented later.

An IBP-Mail transaction takes the following nine steps, also outlined in Figure 3.1:

1. The sender (an IBP-enhanced email client) contacts an IBP-NT agent.

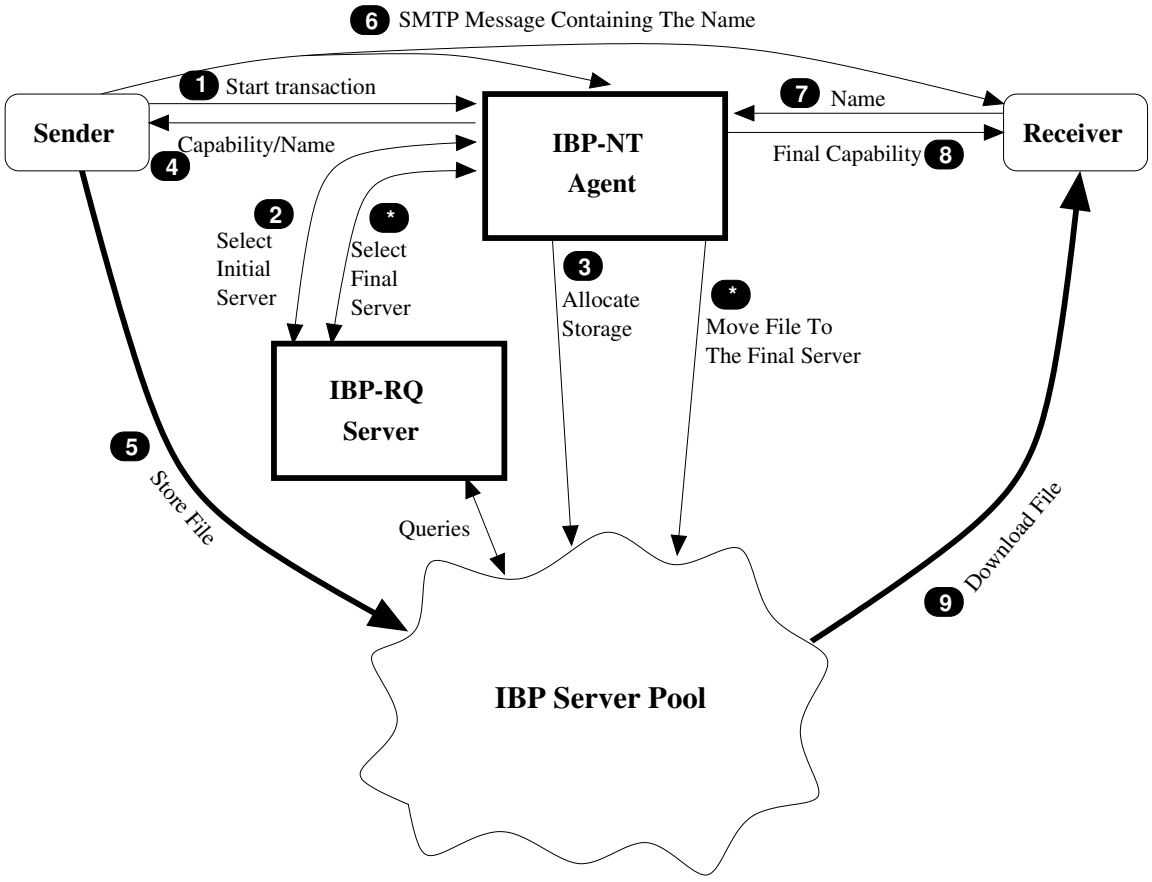


Figure 3.1: An IBP-Mail Transaction

In the preliminary implementation of IBP-mail, there is only one such agent, but multiple agents are possible. The size of the raw binary file is communicated to the agent.

2. The IBP-NT agent queries the IBP-RQ server to list appropriate IBP depots from the server pool that can store the file. The query criteria can include proximity to the sender host, server load levels, server reliability, and/or a combination of the three as well as other server characteristics available to the IBP-RQ server. In the initial implementation of IBP-Mail, available storage available on IBP servers and proximity to the sender are considered.

3. The IBP-NT agent allocates storage for the file on an IBP depot selected from a list returned by the query to the IBP-RQ server, and receives the IBP access capabilities for the newly allocated storage.

4. The IBP-NT agent creates a location independent name for the file and returns that name, plus the write capability to the newly allocated IBP storage area, to the sender.

5. The sender stores the file directly into the IBP depot. The duration of this step depends, among other things, on proximity of the chosen IBP

depot to the sender.

6. The sender sends a standard SMTP mail message to the receiver that includes the location independent name of the file assigned by the IBP-NT agent. At the same time, the sender informs the agent that the file has been written to the IBP server.

7. After receiving the SMTP message, the receiver presents the name to the IBP-NT agent.

8. The IBP-NT agent returns the read and manage capabilities of the file to the receiver. Those capabilities correspond to the *current* location of the file in the IBP depots server pool. Depending on the elapsed time and availability of storage on different IBP depots, this location may correspond to the original IBP depot used by the sender, or another IBP depot closer to the receiver.

9. The receiver downloads the file from the IBP depot, and may delete it if desired, by decrementing the file's read reference count. Note that if a file is shared by multiple recipients, the agent may increment its reference count to equal the number of recipients, and then the file will be deleted only after each recipient has decremented the reference

count (or when the sender-assigned time limit expires).

There are two steps in Figure 3.1 labeled with an asterisk. These may be performed by the IBP-NT agent after the sender stores the file, if the agent determines that it may be able to move the file close to the receiver. If enough time passes between steps 6 and 7 (due to the receiver not reading his or her email instantly), then this time may be used by the agent to move the file close to the receiver(s), thereby improving the time for downloading. If a receiver tries to download the file before it has been copied, it may do so from the original IBP server, with reduced performance.

It should be also noted that the IBP-NT can implement various optimization schemes to accommodate more complex content delivery scenarios. For example, for multi-recipient messages, the agent may elect to replicate the content on several IBP depots that reduce time-to-download for various recipients.

In the following section, we present the details of the various components of the IBP-Mail system.

3.3.1 The IBP registration and query (IBP-RQ) server

The IBP-RQ server provides basic directory registration and query services. In the IBP-Mail system, those services are used by the IBP-NT agent to determine data routing options available to any IBP-Mail attachment in transit. The directory part of the server is a two-level structure, with *groups* containing *hosts* (IBP depots). A *host* can belong to more than one *group* at the same time. This design allows a single IBP depot to be simultaneously used in the logistical networking of more than one IBP application.

The directory functionality embedded in the IBP-RQ server is not much different from other directory technologies, e.g. LDAP [82]). However, the integration of such functionality with the query aspect of the server is what sets this design apart. The query aspect allows clients to submit queries based not only on IBP server characteristics (e.g. total available storage, maximum storage lifetime, . . etc.), but also on networking aspect of the pool of IBP depots and their relationship with other hosts. As such, the IBP-RQ server needs to integrate a dynamic networking querying subsystem that allows it to perform such queries on demand. In the initial implementation of IBP-RQ server, we used Sonar [75]. Several other tools can also be used to provide similar functionality, e.g. the Network Weather Ser-

vice (NWS) [81]. It should be noted that the IBP-RQ server's functionality has been extended and incorporated into the L-Bone (Logistical Backbone) layer. This layer provide more general IBP depots registration and discovery services that are accessible to a wide variety of applications.

3.3.2 The IBP naming and transport (IBP-NT) agent

The IBP-NT agent encapsulates the core functionality of the IBP-Mail system. It implements the user-level routing and data staging operation that distinguish IBP-Mail from traditional content-delivery mechanisms. As the name implies, the IBP-NT agent provides naming and routing services to an IBP-Mail attachment in transit. This service can be provided by a distributed collection of independent agents, or using a single agent (as is the case in the initial IBP-Mail prototype implementation).

As an IBP-Mail client initiate a transaction, it contacts an IBP-NT agent, providing information that characterize the the sending side of the transaction. This information include data size and sender location (host name). The agent then uses its internal metrics to locate an appropriate IBP storage depot for the initial staging of the data. In the initial IBP-Mail prototype, these metrics include the network proximity of the target IBP depot to the sender (determined via the SONAR DNS metric) and available free storage

on IBP depots. Such metrics can be extended to incorporate other properties such as IBP server load levels and reliability, without requiring changes to the IBP-Mail client.

The agent uses a co-located IBP server (or one at close proximity to the agent) to store the access capabilities for the data in transit. This storage area (henceforth called the *name file*) serves as an indirect pointer to the actual attachment as it moves through the IBP depot collection. The read capability to the name file is returned to the sender, along with the write capability to the storage area that has been selected by the agent as the initial data staging point. The sender stores the attachment into the storage area selected by the agent. The read capability to the name file is then incorporated into the STMP message, and is used as a *location independent name* of the attachment. The sender sends the SMTP message *after* successfully staging the attachment at the IBP server. This avoids the need to re-send the SMTP message should a failure occur during initial data staging.

The *transport* phase of the IBP mail transaction is initiated by the agent after being notified (by the sender) of the successful data staging. Along with this notification, the agent is informed of the location of the receiver(s) of the email message. The agent then attempts to move the attachment closer to the recipient(s) (using third party **IBP_copy()**. After each successful copy,

the agent updates the name file with the access capabilities to the new location of the attachment, and deletes the storage area holding the initial data file.

When the receiver presents the name embedded into the SMTP message to the agent, the agent retrieves the read and manage capabilities for the current location of the data attachment from the name file. These capabilities are then returned to the receiver who can use the read capability to read the file, and the manage capability to delete it.

The use of IBP for the name file adds flexibility to the IBP-Mail system. With this structure, it is possible to have multiple agents manage the transport, to have agents restart upon failure without losing information, and even to have the receiver find the location of the file without contacting the agent.

The time-limited allocation property of IBP storage is used to effectively facilitate the resource sharing aspect of IBP-Mail. In addition, it addresses the resource leakage problem in the case of agent failure or lost email. Additionally, it frees the sender and receiver from having to explicitly delete the file. One ramification of this is that there is a new failure mode for mail – time-limit expiration on the data file. There are several ways to address this mode – send warning mail or error reporting mail back to the sender,

send warning mail to the receiver, allow the sender to extend the time limit, or simply delete the file. The IBP-NT agent's functionality has been generalized and incorporated into the Logistical Runtime system (LoRS), a set of tools that allow users to create, manipulate and use network files implemented on top of IBP.

3.3.3 User interface to IBP-Mail system

As part of the initial IBP-Mail prototype implementation, two flavors of mail clients were developed to handle the sender side interaction with the various IBP subsystems. The command line interface (similar to the `mpack`[1]. program) encodes the IBP related information (mainly the location independent name of the attachment) as a special MIME type attachment that is sent along with the traditional SMTP email message. The other client uses a CGI enabled web page to accomplish the same task (this later form allows for wider access and easier experimentation with the system.)

To facilitate processing at the receiver side, the processing required to retrieve an IBP attachment is offloaded to a CGI script accessed through a web server co-located with the IBP agent. The recipient receives a web link that incorporates the name of the transaction. Clicking on this web

[1] `Mpack` may be obtained at `ftp://ftp.andrew.cmu.edu/pub/mpack/`

link invokes the CGI script that performs the name lookup and IBP retrieval operations, and (transparently) redirects the receiver to the current location of the IBP attachment for final delivery.

It should be noted that the use of the web for both sending and receiving introduces certain inefficiencies into the IBP-Mail system. On the sender side, The web interface requires the user to upload the attachment to the web server, which then acts as proxy sender for the email message. This introduces an extraneous data movement as the attachment is moved from the sender's host to the web server, and from there to the initial IBP staging server. Similarly on the receiver side, the use of the web requires data to be moved from the the IBP server holding the data to the web server, and from there to the receiver's machine. These extraneous data movements would be eliminated through the use of IBP-aware mail clients on both ends of the data transaction. However, the use of the web interface eliminates the need for users installing and/or configuring new mail clients to handle IBP-Mail attachments.

3.4 A performance example

In this section we present a simple example that illustrates the use of IBP-Mail to send a relatively large file from the University of Tennessee to Princeton University. We show how performance of IBP-Mail based solution compares with alternative means usually deployed to accomplish the same task. We use a 6 MB file as the test case for transfer in all scenarios. Results reported below are the averages of at least three trial runs on non-dedicated servers and networks performed in the spring of 1999.

3.4.1 Mailing the file

Attempts to mail the file, using either MIME or `uuencode` fail because Princeton's mail daemon, like most, restricts the size of incoming mail. In Princeton's case messages over 6,000,000 bytes are rejected, meaning that this file, which becomes 12.2 MB as a MIME attachment and 12.4 MB with `uuencode` cannot be mailed. The file can be split into three parts and mailed, but this defeats the purpose of limiting the size of incoming mail.

3.4.2 Using FTP

Uploading the file to the ftp server housed at the department of Computer Science, University of Tennessee took 2 minutes and 36 seconds using NFS. Downloading the file from Princeton using anonymous ftp access took 3 minutes. It should be noted that at test time, the ftp server at the University of Tennessee ran on a relatively low-end SPARCstation-2 running SunOS 4.1.4.

3.4.3 Using HTTP

Uploading the file to the web server was accomplished by setting a *soft link* to the original file location. This operation takes no noticeable time. It should be noted that such setup is not always possible, as many organizations separate their web content from other storage, mainly for security reasons. In such a typical setup, uploading time would be dependent on networking infrastructure and the capabilities of the machines involved. At test time, Tennessee's web server was a relatively high end SPARCserver-1000 running Solaris 5.5.1. Time to download the file from Princeton (using the web browser `lynx` for timing) was 29 seconds.

3.4.4 Uploading the file

Uploading the file was not feasible, because the we did not have access to any writable FTP server at Princeton.

3.4.5 IBP-Mail

To test the performance of IBP-Mail, we set up two IBP servers: one that was part of the Internet 2 Distributed Storage Infrastructure (I2-DSI) located at Tennessee, and another DEC Alpha server at Princeton. Storing the file to I2-DSI machine takes one second. Performing a remote copy from the I2-DSI IBP server to the Princeton IBP server takes 46 seconds. Retrieving the file at Princeton takes 10 seconds from the IBP server at Princeton, and 57 seconds from the I2-DSI machine.

3.4.6 Comparison

A comparison of the solutions to mailing the file is shown in Table 3.1. The times are given in *min:sec*. In terms of time spent by the sender, uploading the file to the HTTP server and using IBP-Mail are roughly equal. In terms of time spent by the receiver, if the receiver reads the mail as soon as it is sent and tries to retrieve the file, the HTTP transaction is quicker than

Table 3.1: Comparison of mailing solutions

	FTP	HTTP	IBP-Mail
Time to send	2:36	0:00	0:01
Time to retrieve instantly	3:00	0:27	0:57
Time to retrieve later	3:00	0:27	0:10
Manual deletion required	Yes	Yes	No
Data protected	No	No	Yes

downloading from the I2-DSI machine. However, if the receiver does not read the mail until the file has been moved to Princeton (roughly a minute after it has been sent), then it takes only 10 seconds to download. Thus, in the typical case, where the receiver does not read mail instantly, IBP-Mail outperforms uploading the mail to the HTTP server. Additionally, in this typical case, there will be less variability in the performance of the download, since the file is on a local network with IBP-Mail, instead of on a remote HTTP server.

The last two lines of Table 3.1 pertain to ease-of-use and security. With FTP and HTTP, the sender must manually delete the file when she is certain that the receiver has retrieved it. With IBP-Mail, the file is automatically deleted either by the receiver or by the time-limit on the file. The "data protected" line concerns who can actually read the file. With anonymous

FTP, anyone who can connect to the server may discover and read the file. With HTTP, anyone can read the file, but discovery may be more difficult if the sender arranges that outside users cannot list the directory containing the file. However, since the directory must be world-readable for the web server to serve it, all users at Tennessee are able to find and read the file. With IBP-Mail, anyone can read the file, but only if they have access to the read capability. Since there are no directory semantics exported by IBP, the likelihood of someone discovering the read capability is extremely small. Only the owner of the IBP server process has access to the file.

3.5 Related work

Mail systems have been around for decades; however most mail systems work within the structure of SMTP mail. The Grapevine project [21, 72] developed at Xerox PARC in the early 1980's was a distributed mail handling system that provided functionality beyond simple point-to-point delivery of email messages. Grapevine, however, required a substantial administrative overhead as it handled many features outside mail transport (e.g. user registration and authentication). It was assumed that servers participating in a Grapevine system were under control of a single administrative entity

(or several closely cooperating entities). It is extremely difficult in today's Internet to satisfy this requirement. IBP-Mail assumes no administrative intervention beyond setting up the various servers that are part of the system, and requires no special privileges for its setup and/or operation.

A more recent project is Porcupine from the University of Washington [60, 69]. Porcupine uses clusters of computers to handle mail traffic that can be as much as a billion messages a day. As such, Porcupine is appropriate for organizations that have huge amounts of incoming mail. Like Grapevine, Porcupine works within a single administrative domain, and does not address mail between arbitrary and perhaps unrelated entities on the Internet.

The *External-Body* subtype [43] of the MIME protocol may be used to affect the delivery of an email attachment through a receiver-initiated action. Through the use of this subtype, the sender specifies an access protocol that is recognized by the receiver and the proper protocol handlers are used to access the actual body of the message. The use of this technique requires the sender to pre-stage the message body at a storage site accessible through an access protocol (e.g. FTP or HTTP). While this technique works well for objects that are already accessible through the protocol in question, it requires additional administrative overhead for those objects that are intended

to be exchanged only between the sender and the receiver. This overhead stems from the need to limit access to the object to the receiver(s), the need to detect when the object has been retrieved by the receiver to revoke it's protocol access (or remove it physically from permanent storage), or the need to stage the object at a protocol-enabled server.

The underlying concepts of IBP-Mail, embodied into the IBP-RQ server and the IBP-NT agent, have been extended and incorporated into a more general logistical network framework. The network storage stack [14, 57] combines IBP with higher level abstractions that facilitate applications' use of collection of IBP depots. The two main abstractions at the core of the network storage stack are the L-Bone (Logistical Backbone), a distributed runtime layer that allows client to perform IBP depots discovery. and the exNode, a data structure for aggregation, analogous to the Unix inode. 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 that may be used like a file. These two abstractions provide the basis upon which the Logistical Runtime System (LoRS) is built to provide applications with a high level access to logistical networking functionality.

IBP has been explored as the underlying infrastructure for the delivery of different types of content [9]. In [18], IBP is used as the basis for a reliable

multicast infrastructure in heterogeneous networks. This infrastructure is built on top of the LoRS, effectively building an end-to-end model on top of IBP [16]. The use of IBP and the exNode layer for the streamed delivery of MPEG-2 video files is explored in [11].

Chapter 4

Wide area data staging in task farming applications using IBP

4.1 Introduction

The computational grid [40] has recently evolved into a powerful paradigm of the aggregation and efficient utilization of distributed computing, networking, and storage resources. Grid infrastructure projects, such as Globus [39], Legion [45, 50], NetSolve [26], Condor [37, 76], Ninf [54, 73] and EveryWare [80] endeavor to provide the tools and user interfaces that facilitate efficient access to wide area distributed resources. The role of storage in the grid has been at the center of many of these computational grid efforts. By definition, the computational grid encompasses resources that span a

wide geographical (and network) distance. As such, the issue of data locality plays an important role in enhancing the performance of distributed applications running on the grid. More recently, research into grids targeted primarily at managing the location and distribution of data has been realized into what has become known as *data grids* [32].

Locality has always been an important element in reducing data access overhead, whether at the processor level or in a distributed setting. While it is generally accepted that staging data near where it is consumed (caching) improves general application performance, the nature of such an improvement depends on the type of the application and on its ability to effectively use the available data.

Parameter sweep (or farming) applications compose a class of applications with attractive scalability properties, which renders them suitable for grid implementation. These embarrassingly parallel applications divide a (potentially huge) parameter space into regions. A worker application is then assigned the task of searching one such region for points that match the search criteria set by the application. A sub-class of such applications are *data independent parameter sweeps*, where the input data set is shared between all workers. The structural properties of this sub-class suggest the use of compute-side data staging to reduce input data fetching communica-

tion delay. The resulting reduction in overhead should increase application throughput (expressed in terms of the number of regions searched per unit time).

In this chapter, we study the staging behavior of wide-area data-independent parameter sweep applications. We base our computing model on NetSolve [26], and use IBP as the underlying staging storage model. We first derive an analytical model for predicting performance in such systems, and then run experiments in a wide-area setting to determine how well the model fares in a real implementation. Finally, we show results of a simulation of these systems so that a wider class of parameters can be analyzed.

4.2 Parameter sweep applications

We focus on parameter sweep, or "farming," applications, which have been the focus of much initial work on grid computing [3, 13, 27, 28, 74]. As described above, these are parallel applications where little or no communication is performed between sub tasks in a larger computational process. An example is MCell, a microphysiology application that uses 3-D Monte-Carlo simulation to study molecular bio-chemical interactions within living cells [28]. In fact, almost all Monte-Carlo simulations belong to this class

of applications.

The applications that we study have the following characteristics:

- **Large input data set.** The input data set is large relative to any control parameters that determine the manner in which the data set is processed.

- **Large number of tasks.** The number of tasks is large and is not usually known in advance. The application sweeps through a (possibly infinite) parameter space defined through the control parameters. Task throughput is the paramount measure of overall application performance.

- **Task independence.** Each task is completely defined through its control data set and the global input data set. No dependencies exist between tasks in the parameter sweep application.

4.3 Computing model

We adopt the "client-agent-server" computing model supported by NetSolve [26]. In this model, a client submits problems to computational servers through an intermediate agent, which acts as a resource broker and perfor-

mance monitor for all computational servers that are registered with this agent. The client running the application queries the agent for a list of servers capable of solving the problem at hand. The client then proceeds to submit instances of the parameter sweep application to servers in that list until no servers are left unused. Clients submit tasks to the servers in the form of an asynchronous RPC call that fully defines the computational task at hand. The client then proceeds to monitor completion of tasks on active servers, submitting more tasks to the server(s) that complete their tasks.

The introduction of staging into this environment leads to the differentiation of the control and data streams within the application. Without staging, the client marshals all inputs to the computational problem to the server whenever an instance of the problem is instantiated. With staging, the input data is prepositioned on a storage server, perhaps near the computational servers, and a pointer to the data is sent to the servers on task instantiation. The servers retrieve the data from the storage after their tasks are instantiated. This separates the path of the control data from that of the (larger) input data as seen in Figure 4.1. IBP provides a flexible and efficient platform for the staging storage in this setup.

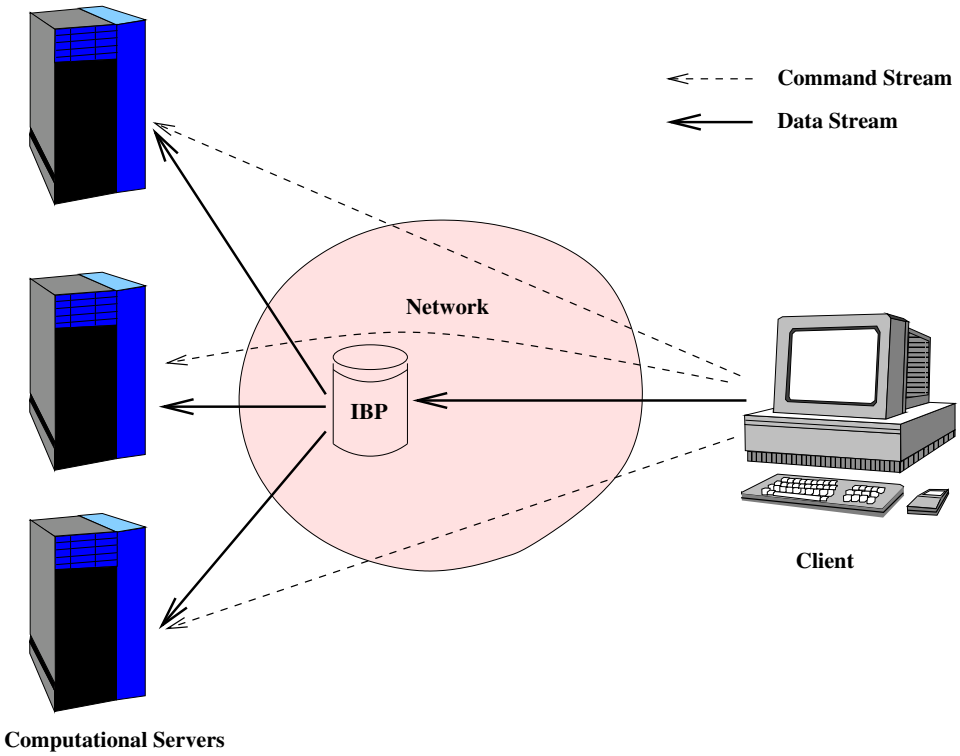


Figure 4.1: Farming model with an IBP Staging Point

4.3.1 Application characteristics

To analyze the effect staging has on the throughput in a parameter sweep application, we make the following assumptions:

- Pure execution time of individual tasks on computational servers is constant. Moreover, each server executes one task at a time.
- Tasks are introduced into the computational environment one at a time. The reasoning behind this assumption is twofold. The simultaneous introduction of tasks from the client side to multiple computational servers imposes heavy demands on (possibly limited) client-side link bandwidth for problems involving large data sets. Such usage of available bandwidth may not be acceptable to many organizations. The second reason is that in some environments (e.g. NetSolve), the introduction of a problem instance into the environment alters the configuration in such a way as to affect the decisions involved in assigning future instances to servers. While devising a scheme for the simultaneous scheduling of multiple problem instances is feasible, it is beyond the scope of this research.
- No server failures occur during computations.

- The output data size is negligible. It follows that no significant time elapses between a task terminating on a server and the client's recognition of this event and the launch of another instance on the same server.

4.3.2 Application throughput model

In analyzing the throughput of parameter sweep applications, we define the following variables:

- T_f: Task forking time. This is the time to select a computational server and start a process on it. It includes network latencies and protocol overhead, but not the time to transmit input data.
- T_t: Input data transmission time.
- T_c: Task computational time, once the input has been received.
- T: A sufficiently large time interval used to measure throughput.
- P: Total number of identical computational servers.
- P_e: Effective number of computational servers.
- N: Number of completed tasks in T seconds.

With no staging, tasks are started as depicted in Figure 4.2(a). The task turnaround time is $(T_f+T_t+T_c)$ *Sec/Task*. Since a new task cannot be started until all data from the previous task has been transmitted, the task initiation

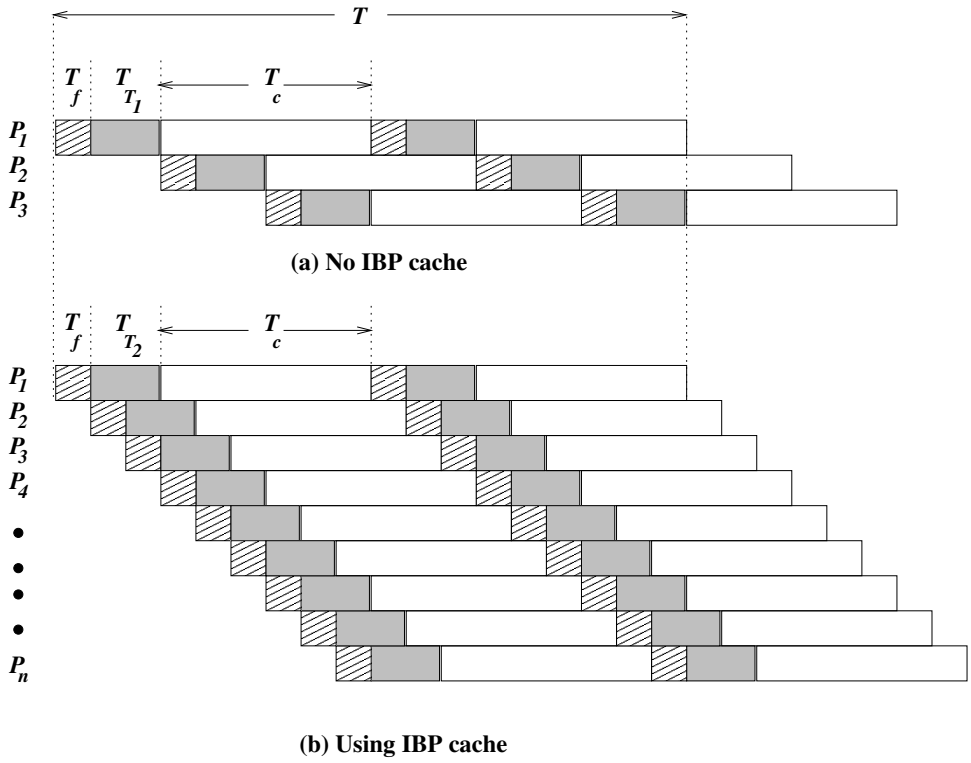


Figure 4.2: Task launching in parameter sweeps

rate is $1/(T_f + T_t)$ *Tasks/Sec.*. It follows that, the maximum number of tasks that can be started before the first task terminates is given by

$$P_e = (T_f + T_t + T_c)\ \frac{1}{T_f + T_t} \quad Tasks/Finished\ task$$

A new task initiated after completion of the first task does not increase the number of simultaneously active servers. By starting this new task on the same server where the recently terminated task was assigned, we get

the scenario depicted in Figure 4.2. It should be noted that this choice is possible because of the assumption that all servers are identical, which makes it immaterial which server gets to process any given task. It follows that, *regardless of the number of available servers*, the maximum number of servers that can be kept *simultaneously* busy (P_e, for "effective" number of servers) depends on $(T_f + T_t)$. In Figure 4.2(a), P_e equals three. Assuming $P_e \leq P$, then

$$\begin{aligned} N &= P_e \frac{T}{T_f + T_t + T_c} - (P_e - 1) \\ &= \frac{T - T_c}{T_f + T_t} \end{aligned} \tag{4.1}$$

With staging, we add primes to all variables. Note that $T'_f = T_f$ and $T'_c = T_c$. However T'_t will differ from T_t, depending on the location of the staging point. As depicted in Figure 4.2(b), a task need not wait for its input before starting, but rather, begins fetching its inputs from the staging point as soon as it is initiated. The result is that tasks are able to fetch their inputs in parallel up to the point where the bandwidth consumed by all tasks in their fetch phase is equal to the capacity of the network link. This increases the effective number of servers (to nine in Figure 4.2(b)), and will also shorten

task turnaround time if $T'_t < T_t$. It follows that

$$P'_e = \frac{T_f + T'_t + T_c}{T_f} \quad Tasks/Finished\ task$$

And assuming $P'_e \leq P$, then

$$\begin{aligned} N' &= P'_e \frac{T}{T_f + T'_t + T_c} - (P'_e - 1) \\ &= \frac{T - T'_t - T_c}{T_f} \end{aligned} \tag{4.2}$$

From equations 4.1 and 4.2

$$\frac{N'}{N} = \frac{T - T_c - T'_t}{T - T_c} \frac{T_f + T_t}{T_f} \tag{4.3}$$

For sufficiently large T, $T - T_c \gg T'_t$. Then

$$\frac{N'}{N} \approx 1 + \frac{T_t}{T_f} \tag{4.4}$$

Equation 4.4 shows that the improvement we see from the use of staging does not depend on transmission time (T'_t) up to the point of link saturation.

The staging performance modeled above assumes the number of available servers exceeds the maximum number of effective servers that a client can use, and that the slowdown of input transfer caused by link saturation

is negligible. The improvement in performance is due to the reduction in transmission time and the enhanced parallelism achieved through overlapping task instantiation and data transmission. To isolate the effect of staging, let $P < min(P_e, P'_e)$. In this scenario, we find that

$$N = P \frac{T}{T_f + T_t + T_c} - (P - 1) \tag{4.5}$$

$$N' = P \frac{T}{T_f + T'_t + T_c} - (P - 1) \tag{4.6}$$

For sufficiently large T, $P - 1 \ll P \frac{T}{T_f + T_{T_x} + T_c}$. It follows that

$$\frac{N'}{N} \approx \frac{T_f + T_t + T_c}{T_f + T'_t + T_c} \tag{4.7}$$

For a server side staging point, we have $T'_t \ll T_f + T_c$. Then

$$\frac{N'}{N} \approx 1 + \frac{T_t}{T_f + T_c} \tag{4.8}$$

For a client side staging point, $T'_t = T_t$ and $N'/N \approx 1$. In the vast majority of scientific farming applications, the computational time T_c dominates data transmission time T_t. This is in contrast to web caching applications, where the opposite is generally true. Hence, it can be seen from the above model that as the number of computational servers available decreases from

that number needed to keep the staging pipeline full, the throughput gain advantage through staging diminishes and eventually vanishes with sufficiently long running tasks and/or few servers.

4.3.3 Sharing of bandwidth to the staging point

In the model described earlier, the transmission time of the staged input data set to the computational server(s), T_t', is assumed to be independent of the number of simultaneous connections between the staging point and the computational servers. This assumption, in effect, indicates the availability of separate and independent connections between the staging host and the computational servers. For most real situations, the staging server has only one communication link that is shared among all connections to the computational servers. This means that as more simultaneous connections are made to the staging server, the time to transfer the staged data on each connection is affected. This causes the effective bandwidth BW_e between the staging server and the computational servers to be different from the nominal bandwidth BW of the connecting link. In what follows, we derive a formula for the upper bound of BW_e. We maintain the assumptions that the total number of available computational servers is large enough to keep the client from being idle between tasks. In what follows, we use the same ter-

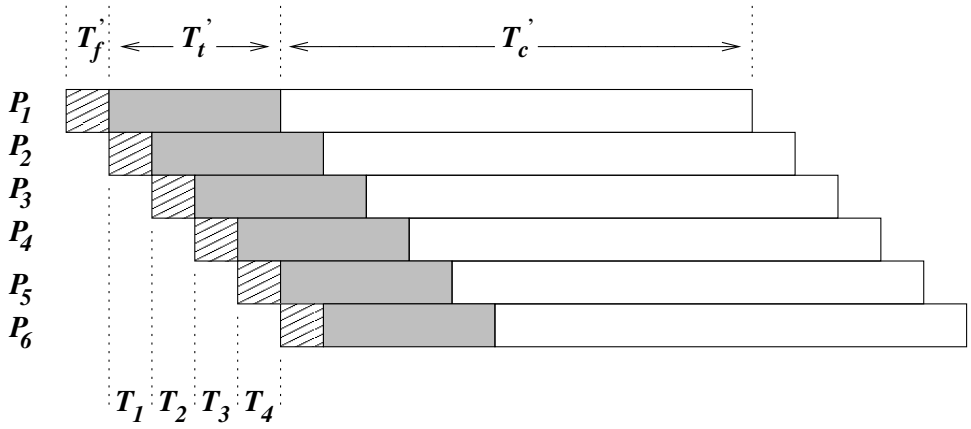


Figure 4.3: Sharing of bandwidth to the staging server

minology defined in section 4.3.2. We assume that the communication protocols used in transmission allow full utilization of any available bandwidth, and fair sharing of available bandwidth between competing transmissions.

Consider Figure 4.3, which shows task instantiating using a staging server. Let S be the size of the data file that is staged and which is transmitted every time a new task is started. For simplicity, assume that the staged data file transmission time T_t' is given by $T_t' = mT_f'$, where T_f' is the task forking overhead incurred by the client, for some integer m. Consider the first task started on P_1, during time interval T_1 (which is equal to T_f'), task on P_1 has exclusive access to the communication link to the staging server. Hence, the amount of data transferred during T_1 is given by $S_1 = BW\ T_f'$.

As a new task is started on P_2 , it shares the communication link to the staging host with the task on P_1 (assuming $S_1 < S$). Hence, the amount of data transferred to P_1 during time interval T_2 is given by $S_2 = (BW\ T'_f)/2$

Similarly for tasks started on P_3 and P_4, each new task reduces the amount of bandwidth available to P_1. This process continues until time interval T_n during which data transmission for P_1 terminates. During time interval T_n, n simultaneous connections share the communication link to the staging server. During time interval T_{n+1}, the task running on processor P_{n+1} starts transferring its instance of the staged data file, thus the communication link continues to be shared among n or more transmissions assuming the number of available computational servers is large enough. All subsequent transmissions will have an effective bandwidth $BW_e \leq BW/n$. The inequality stems from the fact that task started on P_2 will not necessarily finish transferring its input during time interval T_{n+1} (since it started with only half the nominal bandwidth). This allows yet more tasks to start, further reducing the bandwidth available to any one server. Then, we can write

$$\begin{aligned} S &= T'_f\ BW\ +\ T'_f\ \frac{BW}{2}\ +\ \cdots\ +\ T'_f\ \frac{BW}{n} \\ &= T'_f\ BW \sum_{i=1}^{n} \frac{1}{i} \end{aligned} \tag{4.9}$$

Then for $T_t' = T_f'$, we have $S = T_f' \ BW$ and $BW_e = BW$. For $T_t' = 2T_f'$ we have $S = 3/2 \ T_f' \ BW$ and $BW_e \leq BW/2$. For larger values of T_t', we can write

$$S \approx T_f \ BW[ln(n) + \gamma] \tag{4.10}$$

where γ is known as the *Euler-Mascheroni* constant and is approximately equal to 0.5772156649. It has been shown [83] that the error in the above approximation is bound by the formula

$$\frac{1}{2(n+1)} < \sum_{k=1}^{n} \frac{1}{k} - [ln(n) + \gamma] < \frac{1}{2n} \tag{4.11}$$

Note that the approximation in equation 4.10 can be used for all values of n, subject to the error constraints defined in equation 4.11. Solving equation 4.10 for n, we get

$$n \approx e^{\frac{S}{T_f' \ BW} - \gamma} \tag{4.12}$$

equation 4.12 shows that the effective bandwidth connecting the computational servers to the staging point decreases exponentially as the staged data size increases relative to the maximum amount of data that can be transmitted from the staging server in time interval T_f'. If n is not limited by the

number of available servers, BW_e could decrease to the point where virtually no progress is made in the farming application as all instantiated tasks try to simultaneously retrieve their common input data set.

4.4 Experimental results

In this section, we present the results of running a wide area distributed farming application. The goal of this part is to test the validity of the model developed in section 4.3 under real network and server loading conditions (computational servers running in work stealing mode). We use a simple synthetic algorithm that allows us to control the running time of computational tasks through input parameters. Multiple instances of this algorithm were spawned on a network of computational servers from a remote client.

4.4.1 The synthetic algorithm

This simple algorithm is designed to allow user control of the complexity (and consequently running time) of an application. The pseudo-code for the algorithm is shown in Figure 4.4. This algorithm has complexity of the form $\alpha\ C\ n^{\beta}$, where α is an input parameter, C is a constant multiplier specific to the algorithm, n is the input data size, and β is the controlled exponent

```
for i = 1 to alpha step 1
    for a = 1 to n step inc_a
        for b = 1 to n step inc_b
            for c = 1 to n step inc_c
                Perform fixed set of computations
            end for
        end for
    end for
end for
```

Figure 4.4: Synthetic testing algorithm

such that, $\beta \leq 3.0$

By setting $inc_a = n^x$, $inc_b = n^y$, and $inc_c = n^z$, the complexity of the algorithm is found to be $\alpha C n^{(1-x)+(1-y)+(1-z)}$. Simplifying, we get

$$complexity \; = \; \alpha \; C \; n^{3-(x+y+z)}$$

By varying x, y, z such that $0 \leq x, y, z \leq 1$, we can control the complexity of the algorithm from $\alpha \; Cn^3$ to an insignificant constant.

4.4.2 The testing environment

The first set of experiments experiments using the synthetic algorithm were performed using the following setup:

- **Client:** The client is a SUN ULTRA-2 machine running at Princeton University.

- **Servers:** The computational servers are selected from a pool of 41 machines running at the University of Tennessee, Knoxville. Of these, 12 machines are dual 167-MHz UltraSPARC-1 processor machines with 256-MB of memory. The remaining 29 machines have single 143-MHz UltraSPARC processors with 256-MB of memory.

- **NetSolve agent:** The NetSolve agent was launched on one of the computational servers used in the experiment.

- **Staging Point:** The staging point is implemented using an IBP server running on an Internet2 machine dedicated to serving storage to network applications [15]. This machine is housed at the University of Tennessee.

- **Experimental setup:** All reported experiments were performed between the hours of 8:00 AM and 11:00 PM (EDT) during the months of May and June 2000. All components of the experiments were running in a work-stealing mode, with a *nice* value of 15.

The farming interface [27] in NetSolve was used to launch tasks and assign them to computational servers. However, this interface was instrumented to allow for better control over the maximum number of hosts that can be simultaneously used by a client, and to add various tracing and performance measurement code.

4.4.3 Results

In this section, we present results from two experiments. In both experiments, each task implements an instance of the synthetic algorithm presented in section 4.4.1, with computational complexity n^y such that $0.6 \leq y \leq 1.1$ In the first experiment, the number of computational servers (P) is fixed at 35. In the second experiment, P is six. The goal of these experiments is to test the validity of the models developed in section 4.3 as well as the simulator code described later. All runs were conducted for a period T of 3 hours. Input data size for these experiment is 832 KB. Tables 4.1 and 4.2 summarize the results of these experiments.

As the last two columns of each table indicate, the model does a very good task of predicting actual behavior both with and without staging. And as is demonstrated most clearly in Table 4.1, the most notable improvement due to staging is the increase in P_e — roughly a factor of two in Table 4.1.

Table 4.1: Experimental results (1). $P = 35$, $T = 3$ hours

Complexity	T_f (sec)	T_t (sec)	T_t' (sec)	T_c (sec)	T_c' (sec)	P_e		P_e'		N		N'	
						Calc.	Act.	Calc.	Act.	Calc.	Act.	Calc.	Act.
$n^{0.6}$	2.8	7.5	0.2	15.6	9.84	2.5	1.5	4.6	3.4	1050	1047	3895	3758
$n^{0.7}$	3.4	7.8	0.2	37.7	26.81	4.3	3.3	8.9	7.4	963	953	3168	2984
$n^{0.8}$	4.7	6.2	0.2	90.7	82.68	9.3	8.3	18.6	17.3	980	983	2280	2257
$n^{0.9}$	6.6	6.0	0.2	257.2	243.18	21.3	22.4	37.7	33.2	833	851	1478	1462
$n^{1.0}$	6.7	8.0	0.2	737.4	744.80	51.1	33.4	111.7	34.4	469	390	469	452
$n^{1.1}$	7.1	8.5	0.2	1458.8	1471.71	94.5	33.8	207.7	34.5	222	230	222	247

Table 4.2: Experimental results (2). $P = 6$, $T = 3$ hours

Complexity	T_f (sec)	T_t (sec)	T_t' (sec)	T_c (sec)	T_c' (sec)	P_e		P_e'		N		N'	
						Calc.	Act.	Calc.	Act.	Calc.	Act.	Calc.	Act.
$n^{0.6}$	2.6	7.2	0.2	10.9	10.6	2.1	1.1	5.2	4.0	1104	1100	4198	4095
$n^{0.7}$	2.6	7.3	0.2	30.4	29.6	4.0	3.0	12.0	5.4	1075	1071	1989	1978
$n^{0.8}$	2.7	7.7	0.2	89.8	85.0	9.7	5.3	32.3	5.8	642	639	733	736
$n^{0.9}$	2.8	7.7	0.2	257.6	254.8	25.5	5.7	93.0	5.9	237	239	247	248
$n^{1.0}$	2.9	8.1	0.4	757.7	895.7	70.3	5.9	279.0	6.0	79	82	67	69
$n^{1.1}$	3.2	8.5	0.2	1515.8	1560.6	130.0	5.9	484.2	6.0	37	42	36	39

This increase in P_e is seen to be over a factor of three in Table 4.2. However, when there are not enough processors for P_e processors to be kept busy, as in the last two rows of Table 4.1 and the last four rows of Table 4.2, the reduced transmission time due to the input being near the servers has little effect. Below we make a few more comments from the experiments:

- **Forking time T_f:** The average forking time increases as the number of computational servers used increases. The increase stems from the need for the client to negotiate with the NetSolve agent and servers to select a suitable server for every new task. As a result, the average starting time tends to increase as the number of available servers increases due to the dynamic server selection process. In all staging experiments, T_t' was considerably less than T_f; thus no bandwidth sharing effects were observed in the experiments we conducted. The effect of bandwidth sharing is studied in a second set of experiments in section 4.4.4.

- **Transmission time T_t:** In NetSolve, one can only measure total task instantiation time at the client. Without staging, this is $T_f + T_t$. With staging, it is simply T_f, since the data is downloaded from IBP after task instantiation. For that reason, T_t is calculated to be the task in-

stantiation time without staging ($T_f + T_t$), minus the task instantiation time with staging (T_f).

- **Execution Time T_c:** The execution times listed in Tables 4.1 and 4.2 represent the time between the end of the client's task instantiation and the time at which the client detects task completion. As a result, this time may exceed the actual execution time, because the client does not check for task completion while in the midst of forking a new task. As a result, execution time as seen at the client may exceed the actual execution time by ($T_f + T_t$) seconds without staging, and by T_f seconds with staging. This accounts for the differences in T_c and T_c' in the tables.

Finally, in Figure 4.5, we plot the number of completed tasks for each test, both with and without staging. When $P_e < P$, the reduced task forking time of the $P = 6$ tests show better performance. Additionally, when $P_e < P$, staging again shows benefits because it increases P_e. When P_e reaches P, the benefits of staging are far less dramatic.

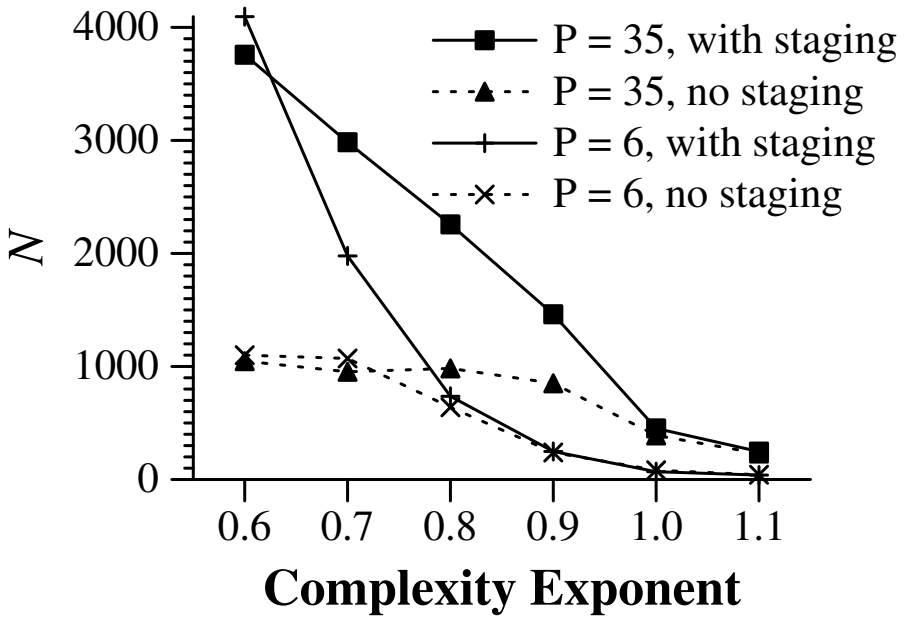


Figure 4.5: Completed tasks in each experiment.

4.4.4 Bandwidth sharing experiments

To verify the bandwidth sharing model, another set of experiments using NetSolve as the underlying architecture was conducted. The experiment studies the effect of variation in the task forking time T_f' on the total task throughput using input data pre-staging via IBP. The experiments used the following setup:

- **Client:** The NetSolve client was a Sun Ultra machine located at Princeton university.

- **Computational Servers:** Computational servers were taken from a pool of 28 Sun Sparc 1 machines at the University of Tennessee, Knoxville.
- **The NetSolve Agent:** The agent was launched on one of the computational servers used in the experiments.
- **The staging point:** The staging point (using IBP) was chosen to be a Sun Sparc-5 workstation accessible through a 10 Mb/Sec NIC located at the university of Tennessee.

The choice of a staging server with a low bandwidth connection to the computational servers was made to allow the sharing of bandwidth to take place for relatively small input data sets and using the observed range of values of T_f and available servers in the computational server pool.

According to the model in section 4.3.3, the effective bandwidth between the staging point and the computational server(s) decreases exponentially with the ratio $S/(T'_f\ BW)$. To test the validity of that model, the remote process forking code on the client side in NetSolve was instrumented to introduce artificial delays that alter the value of T'_f. The tests were conducted using the synthetic algorithm outlined in section 4.4.1, with complexity range of $n^{0.5}$ to $n^{0.8}$. The input data set size was fixed at 9 MB. Table 4.3 shows the results of the test runs, along with results using the

Table 4.3: Effect of bandwidth sharing on task throughput for staged data

Cmplx.	T'_f Delay (sec)	$T'_{f_{min}}$ (sec)	$T'_{f_{ave}}$ (sec)	T'_c (sec)	T'_t (sec)	N'		N'_{fail}	n		P'_e	
						Act.	Calc.		Act.	Calc.	Act.	Calc.
$n^{0.5}$	0	2.0	11.89	33.02	210.87	819	888	37	17	2	22	5
	2	4.0	12.10	31.79	207.67	826	872	30	16	2	20	5
	4	6.0	11.77	27.60	209.77	830	897	49	17	2	23	5
	6	8.0	12.22	27.07	132.48	856	870	7	11	2	13	4
	8	10.0	12.93	25.83	11.15	808	832	0	1	2	3	4
$n^{0.6}$	0	2.0	9.17	64.73	231.18	831	1145	48	18	2	26	10
	2	4.0	10.24	65.67	226.63	820	1026	52	18	2	26	9
	4	6.0	12.43	63.15	143.59	821	852	18	11	2	18	7
	6	8.0	12.50	66.65	122.67	818	848	6	10	2	15	7
	8	10.0	12.61	58.60	11.08	835	851	0	1	2	5	7
$n^{0.7}$	0	2.0	9.71	191.33	135.45	847	1078	6	11	2	26	23
	2	4.0	11.19	189.62	124.47	845	936	3	10	2	25	20
	4	6.0	12.17	194.05	19.08	866	870	0	2	2	18	18
	6	8.0	13.49	196.51	13.52	781	785	0	1	2	16	16
$n^{0.8}$	0	2.0	11.54	703.25	23.70	394	382	1	1	2	27	28
	2	4.0	10.65	685.10	20.62	387	395	0	1	2	27	28
	4	6.0	17.22	671.92	12.55	399	403	0	1	2	27	28
	6	8.0	15.98	682.30	11.21	399	399	0	1	2	27	28

theoretical model developed in section 4.3.3.

The second column in table 4.3 shows the delay introduced into the task forking phase of the computation, while columns 3 and 4 show the minimum and mean observed values of T'_f, respectively. Column 5 shows the average task pure computation time (T'_c) for each setting and column 6 shows the average task input data transfer time T'_t (from staging server to computational server). Columns 7 and 8 show the actual number of completed tasks in a 3 hour duration and the number predicted using the theoretical model, while column 9 shows the number of instantiated tasks that failed due to congestion on the communication link to the staging server. Columns 10 and 11 show the experimental average number of simultaneous connection sharing the communication link to the staging server and the corresponding number estimated using equation 4.12. Columns 12 and 13 show the experimental mean number of simultaneously active computing servers(P'_e) and the model computed counterpart, respectively. As can be seen from table 4.3, the model does a good job in predicting the number of completed tasks, while the model's prediction of (P'_e) and the number of connection sharing the communication link (n) are not as accurate, especially for low complexity (i.e. short duration) tasks. This behavior can be attributed to the following factors:

- **Model quantities** The model estimates a *lower bound* of n, the number of computing servers sharing the communication link to the staging server. This lower bound is typically a loose one, especially in scenarios where the task computational time is significantly less than data transfer time T_t'. This under-estimation could have little effect on the model estimation of the total number of completed tasks for certain combinations of T_f', P, and T_c'. To further illustrate this effect, consider the graphs in Figures 4.6 and 4.7. In these figures, the model prediction of the total number of tasks completed in a three hours time interval, as well as the model estimation of the effective number of computational servers are plotted against n (the number of simultaneous connections that share the bandwidth to the staging server) and for values of T_c' that represent the different complexities used in the experiment. The model results shown in the figures use a computational server pool of size 28, and a bandwidth from computational servers to staging point of 0.74 MB/sec (this is equivalent to the observed bandwidth of the link from the staging server to the computational servers). It can be seen from the graphs in Figures 4.6 and 4.7 that as the task forking time (T_f') used in the model increases, there develops a range of values of n where the total task throughput remains roughly un-

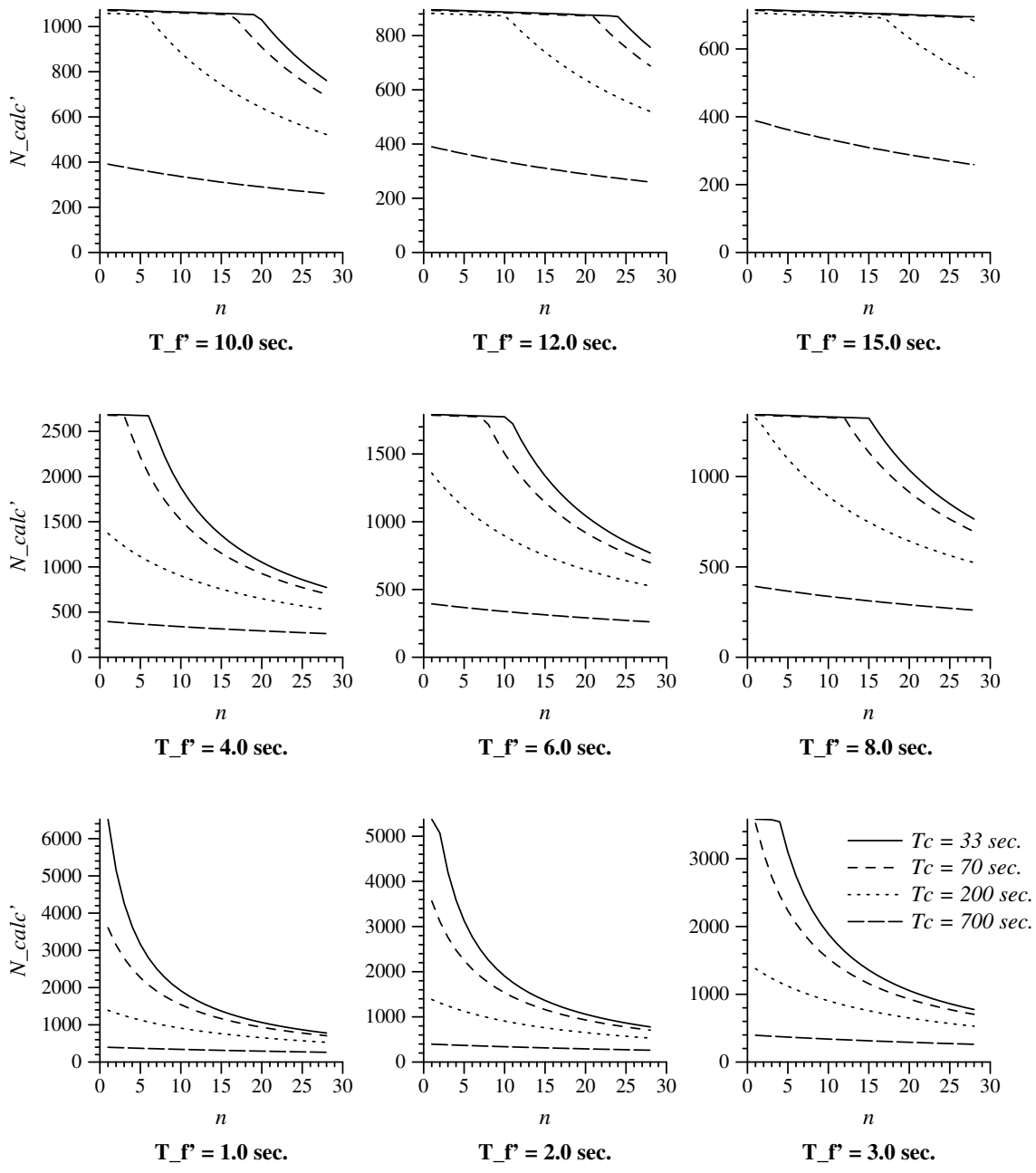


Figure 4.6: Effect of variation in number of shared connections on model predicted number of completed tasks.

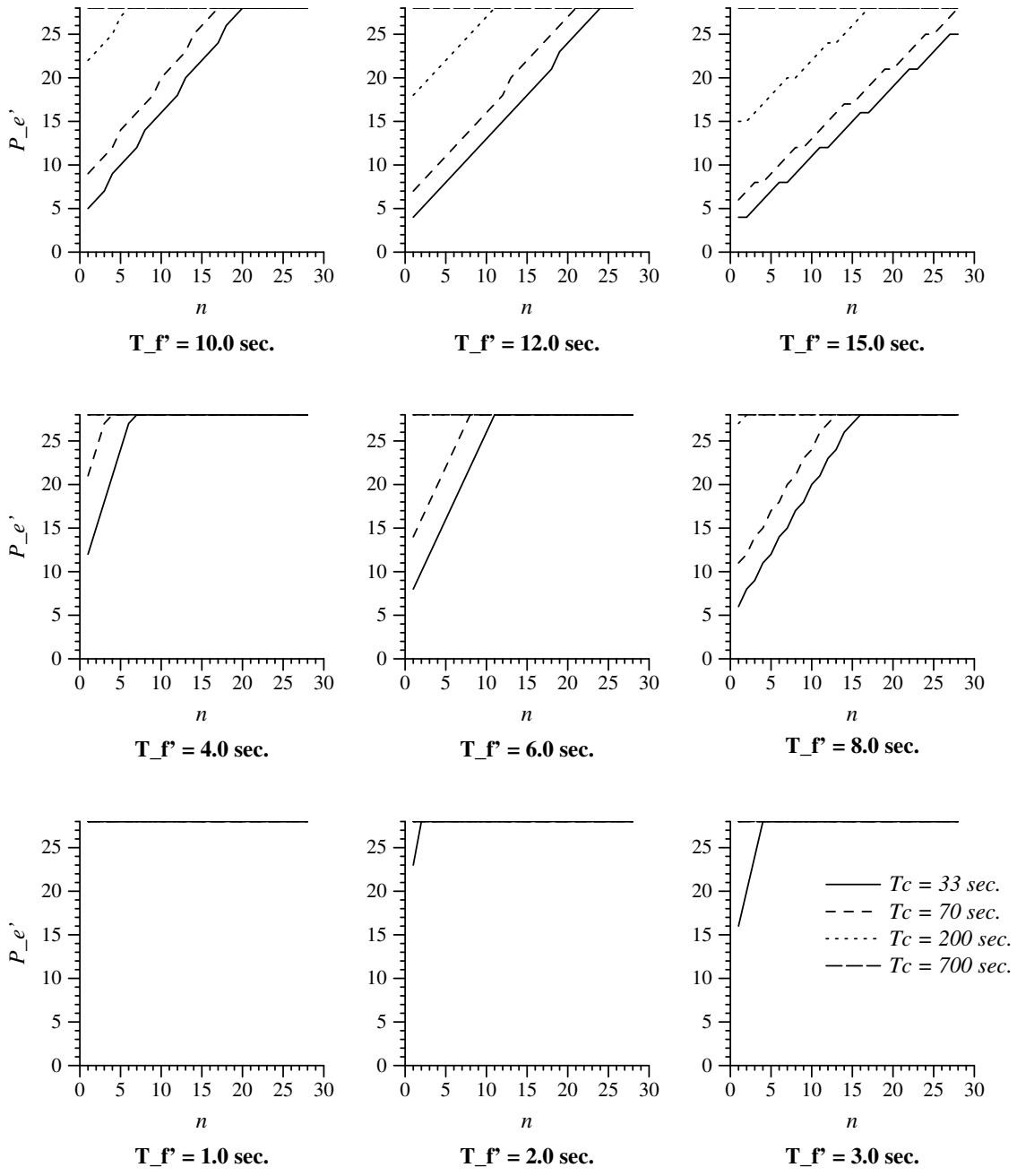


Figure 4.7: Effect of variation in number of shared connections on model predicted effective number of computational servers.

changed, even as the effective number of computational servers (P'_e) increases. Hence, under-estimating n in this region would have a greater impact on the estimates for P'_e rather than on N'.

- **Variation in T'_f** While the model assumes a constant value of the forking time T'_f, the non-dedicated nature of the experimental setup and the dynamic nature of the task forking process result in significant variations in T'_f. As can be seen from the table, the change in data transfer time is affected more by change in the *minimum* value of T'_f rather than its mean value, which is used in the model computation. This can be seen clearly in results for a complexity of $n^{0.5}$, where the introduction of a delay of eight seconds into the forking phase increases the mean value of T'_f by only 1.04 seconds, while the minimum value increases by an amount corresponding to the introduced delay. This suggests that for improved accuracy, a more elaborate model that accounts for variation in T'_f is needed.

The dependency of the model on T'_f can be seen in graphs in Figure 4.8. The graphs show the model calculation of the number of links sharing the bandwidth to the staging server (n), the effective server pool utilization (P'_e), and the total task throughput in three hours (N') for different values of T'_c

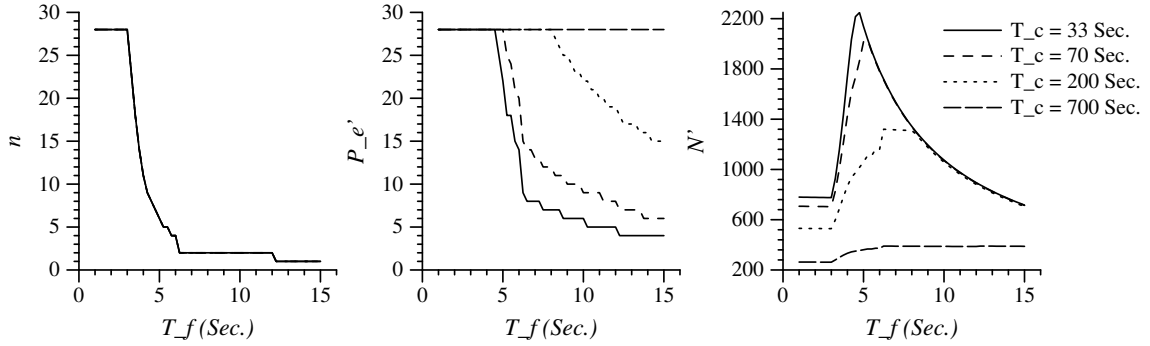


Figure 4.8: Effect of variation in T_f on model predicted quantities.

representative of the complexities used in the experiment. The graphs are obtained using input data set size of 9 MB and a bandwidth from staging point to computational server of 0.74 MB/sec. As can be seen from the graphs for n, the exponential dependency of n on quantity $\frac{S}{T'_f\ BW}$ leads to a rapid drop in the calculated value of n as T'_f increases. This effect contributes to the error in model prediction of n in a non-dedicated environment, as small variation in T_f leads to significant change in n. From the figure, we can see that for small values of T'_f, the bandwidth sharing effect limits the total task throughput (especially for short running tasks), even though the model predicts full utilization of existing pool of computational servers. As T'_f increases, the effect of bandwidth sharing starts to diminish and the total task throughput starts to increase. This increase continues un-

til the increase in T'_f starts to affect the effective number of computational servers utilized, P'_e, which in turn reduces the total task throughput N'. It should be noted that in the case of long running tasks, this later phase will take place at relatively large values of T'_f.

It should be noted that other factors affect the accuracy of the aforementioned bandwidth sharing model. The model developed in section 4.3.3 provide a model for bandwidth sharing behavior at the start of a task farming application. The behavior in subsequent stages depend mainly on the server pool utilization, and the task computational time T'_c. These two parameters determine *when* a new task is instantiated to a computational server and whether this new task will share the bandwidth to the staging server with a previously instantiated task. The model is also highly sensitive to variations in the different parameters it relies on, which occur in non-dedicated environments. The model however provides a first order approximation of the bandwidth sharing behavior and presents some insight as to when such an effect should be accounted for in a distributed task farming application that relies on pre-staging of input data.

4.5 Simulation results

In this section we report further results that we obtained through simulation. The simulator accepts the following parameters:

- Probability distributions for T_f, T_c, remote bandwidth from the client to servers, and local bandwidth from servers to their staging points.
- The input data size S, the number of servers P, and the application duration T.

It then performs a stochastic simulation of the farming application, and returns the number of tasks completed. We present the results of two sets of simulation runs, the first set studies the performance of the wide area task farming application using staging in comparison with that behavior without staging. The second set of simulation runs looks more closely at the effects of bandwidth sharing on staging application performance.

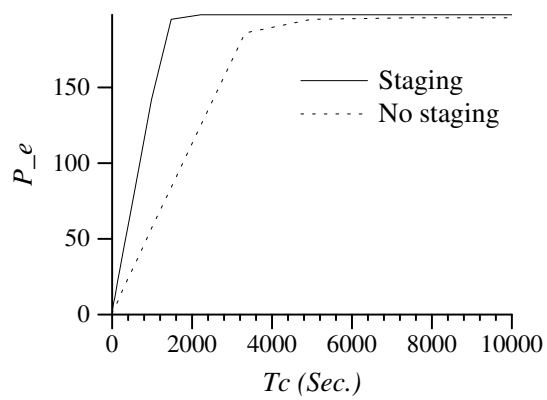
4.5.1 Simulations to study staging effects

For the runs described here, we sample T_c from a uniform distribution in the interval $[\mu - 0.25\ \mu, \mu + 0.25\mu]$, where μ is the mean value of T_c, and varied in these tests. Remote bandwidth is uniformly distributed in the in-

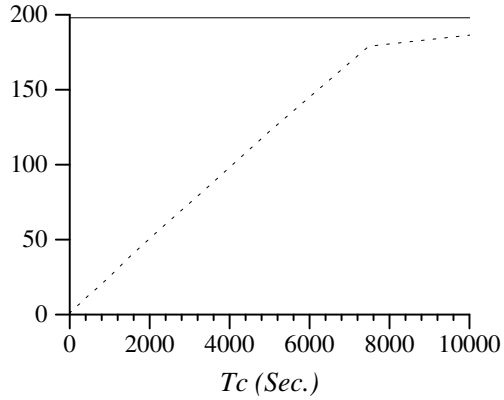
terval $[0.25, 0.75]$ MB per second. Local bandwidth is uniformly distributed in the interval $[0.5, 1.5]$ MB per second. We vary the input data size S between 5 MB and 32 MB and the number of servers P between 40 and 500. Application duration T is fixed at 24 hours.

We used two settings for T_f to simulate two different scenarios. In the first set of results shown in Figure 4.9, we use T_f uniformly distributed in the interval $[3, 7]$. This setting simulates remote dynamic scheduling where significant overhead is used in selecting computational servers. In the second set of results shown in Figure 4.10, we use T_f uniformly distributed in the interval $[0.05, 0.15]$. This simulates a more static scheduling approach where computational servers are selected in a more time-efficient manner (e.g. using round-robin scheduling).

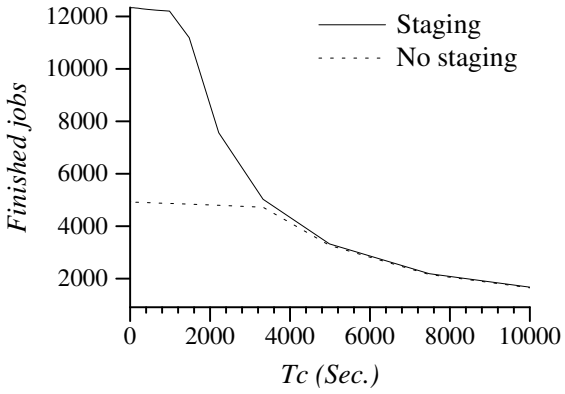
Results shown in Fig 4.9 and 4.10 are the averages obtained over ten independent runs of the simulator. The figures show the results for the throughput ratio (N'/N),the number of finished tasks (N', N), and the effective number of computational servers used (P'_e, P_e). The values chosen represent an environment with decent remote bandwidth (0.5 MByte/Sec) and local bandwidth that is twice that amount. The results reported here will be necessarily different for other bandwidth values, but the model and general conclusions should remain valid.



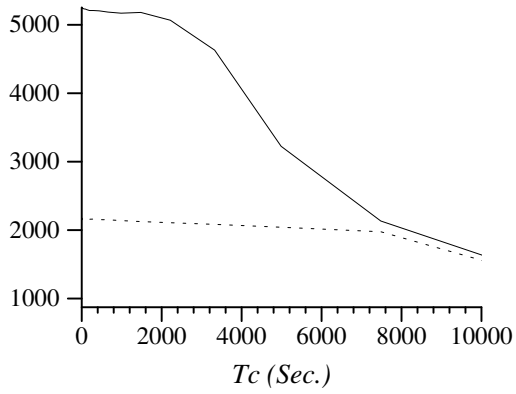
Data Size = 5 MB, P = 200



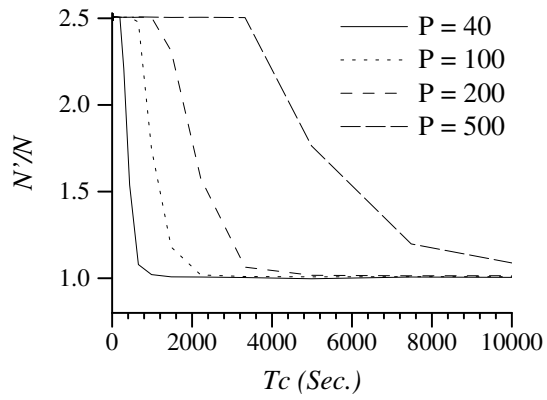
Data Size = 16 MB, P = 200



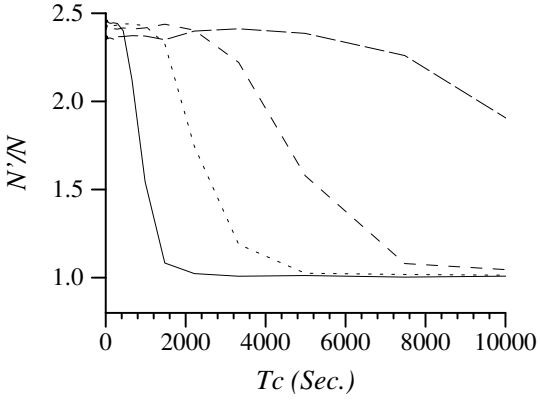
Data Size = 5 MB, P = 200



Data Size = 16 MB, P = 200

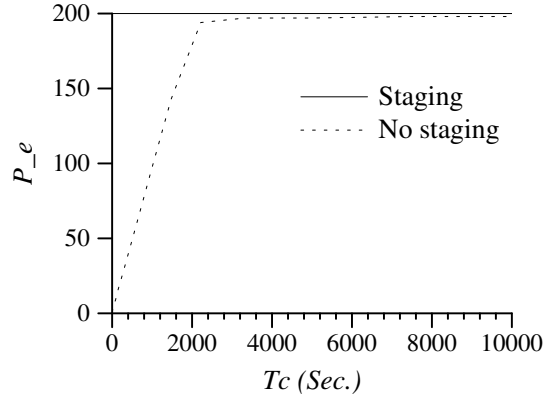


Data Size = 5 MB

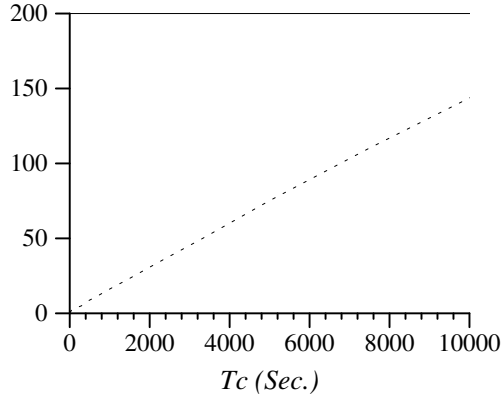


Data Size = 16 MB

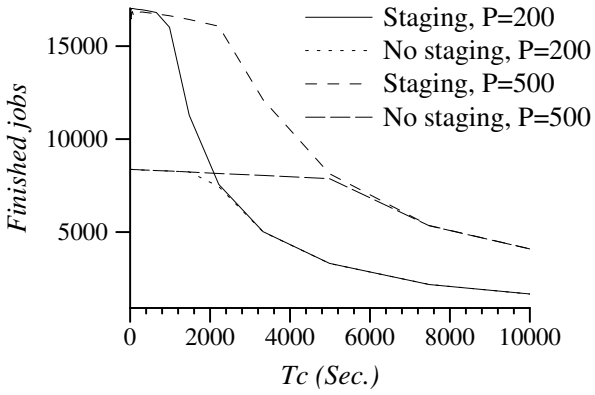
Figure 4.9: Simulation results, $T_f = [3, 7]$ Sec.



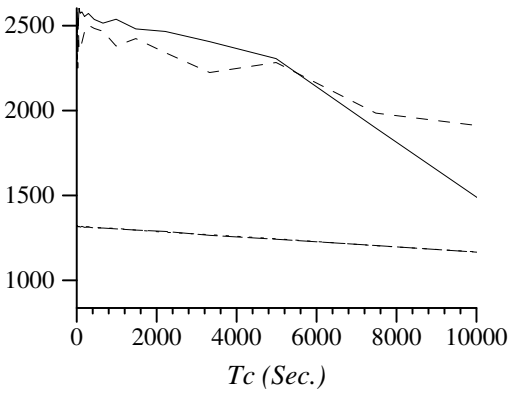
Data Size = 5 MB, P = 200



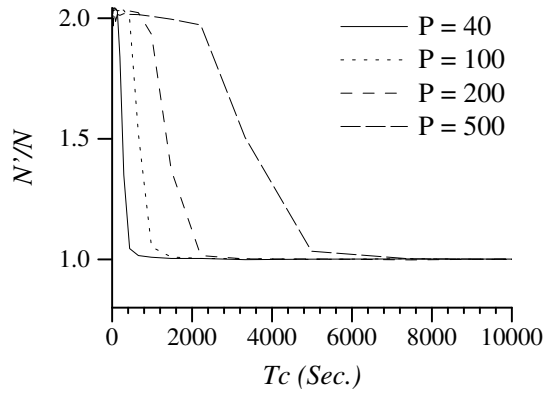
Data Size = 32 MB, P = 200



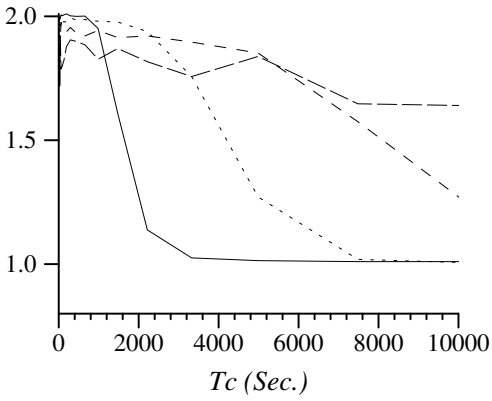
Data Size = 5 MB



Data Size = 32 MB



Data Size = 5 MB



Data Size = 32 MB

Figure 4.10: Simulation results, $T_f = [.05, .15]$ Sec.

In Figure 4.9, the results are shown for input data sizes of 5 MB and 16 MB, and $P = 200$. In these graphs, three different zones can be identified in terms of the individual task computational time T_c:

Zone A This zone extends from $T_c = 0$ to approximately $T_c = 1200$ seconds. In this zone, neither model is able to utilize all available computational servers. The staging architecture is able to utilize more servers due to the reduced total task starting time.

Zone B This zone covers the interval $[1200, 3200]$ for a data size of 5 MB, and $[1200, 8000]$ for a data size of 16 MB. In this zone, the staging architecture is making full use of all available computational servers, while its no-staging counterpart is making partial use of them. It should be noted that the range of T_c for which this condition is true increases as the data size increases.

Zone C This zone covers the interval $T_c \geq 3200$ for a data size of 5 MB, and $T_c \geq 8000$ for a data size of 16 MB. For values of T_c in this zone, both architectures are making full use of all available servers. For sufficiently large running interval T, no significant difference is observed between the total number of finished tasks in both cases. For this region, staging does not improve task throughput.

It can be seen that the precise boundaries between the three zones depend

on the particular configuration of the problem solving environment (various bandwidth values and number of available servers) as well as properties of the problem itself, namely input data size. For instances in zones A and B, staging is beneficial in improving task throughput, while for zone C, such improvement is not observed.

In Figure 4.10, simulation results are shown for a data size of 5 and 32 MB using 200 and 500 computational servers. In this scenario, the effects of bandwidth sharing can be observed for large input data sizes and small values of T_f. It can be seen that for values of T_c less than 5000 seconds, using 500 servers is actually yielding worse performance than the use of 200 computational servers. The increase in transfer time due to the increased number of simultaneous connections increases total task turnaround time to the point that nullifies the benefits gained from the extra computational servers. As T_c increases beyond 5000 seconds, the percentage of time spent transferring data is reduced to the point that we start seeing gains from the use of the extra 300 servers (although the gains are modest in the shown results).

4.5.2 Simulations to study bandwidth sharing behavior

As discussed in section 4.4, the theoretical model for the sharing of bandwidth to the staging point is limited in its accuracy by sensitivity to model parameters as well as its limitation to predicting a lower bound on the number of links that share that bandwidth. In this section, simulation results are presented that further explore the bandwidth sharing effect. The simulations are conducted using a mean value of the bandwidth between the client and the staging server of 0.2 MB/Sec and between the staging server and the computational server(s) of 0.74 MB/Sec. The simulations use a pool of 28 computational servers, input data size of 9 MB, and task computational time that range from 33 seconds to 700 seconds. The results of these simulations are shown in Figure 4.11. In Figure 4.11-a, all simulation parameters are kept fixed at their mean value. In Figure 4.11-b, simulation parameters were sampled from a normal distribution with the mean value specified, and a standard deviation σ equal to $0.255\ \mu$. This choice of standard deviation corresponds to a normal distribution with 0.95% of the distribution in the interval $[\mu - 0.5\mu,\ \mu + 0.5\mu]$. This setup simulates a scenario with medium parameter variation. In Figure 4.11-c, the standard deviation is doubled (to 0.51μ) to simulate an environment with increased variation in parameters.

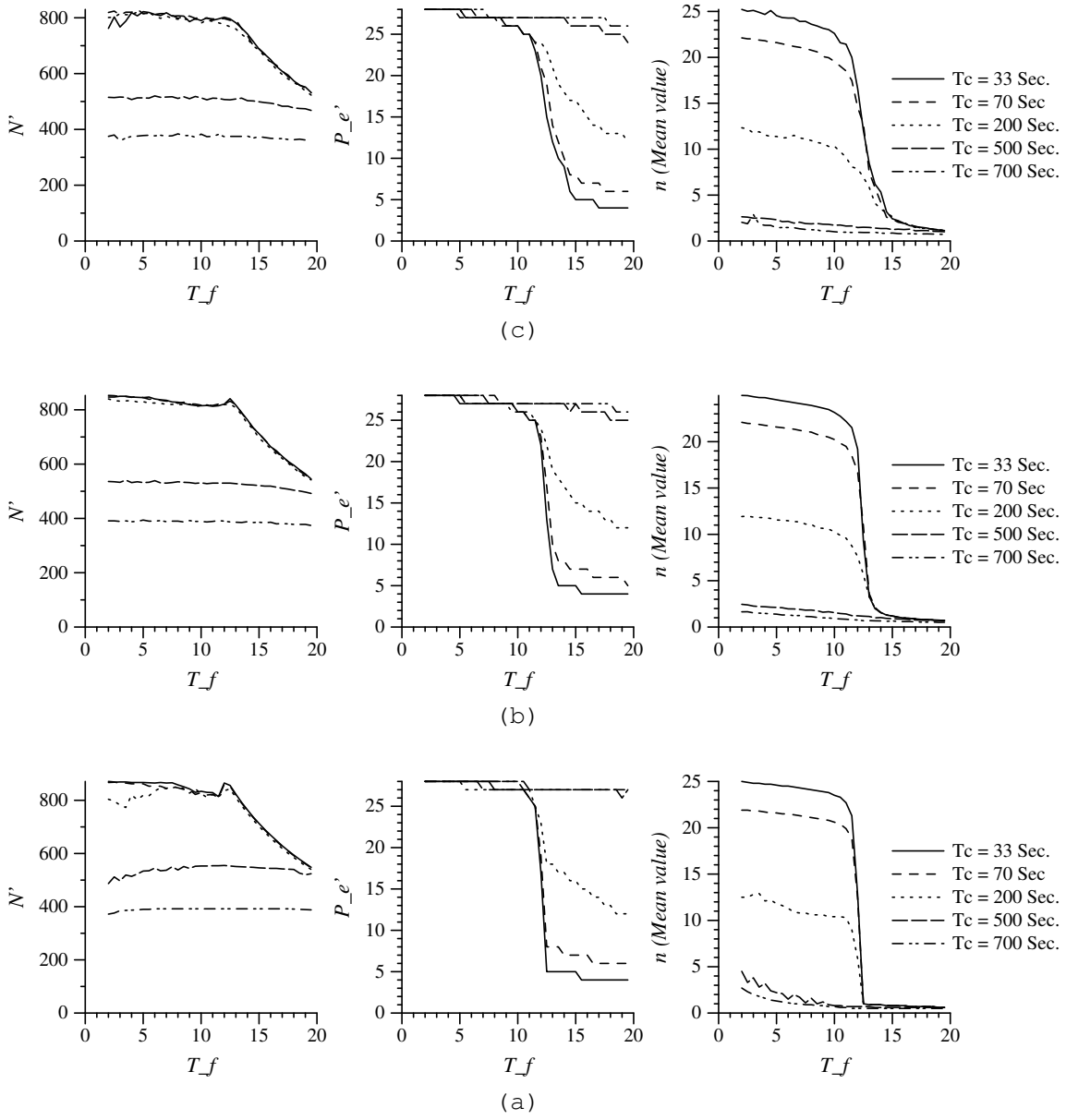


Figure 4.11: Bandwidth sharing simulation results. (a) Fixed parameters. (b) Normal distribution 1 ($\sigma = 0.255\ \mu$) (c) Normal distribution 2 ($\sigma = 0.51\ \mu$)

The results shown are the average of ten independent runs for each data point.

As can be seen from Figure 4.11, the average number of simultaneous transfers that share the link from the staging server to the computational server(s) drop sharply as T'_f approaches 12 seconds. This corresponds to the minimum value of T'_f required to guarantee exclusive utilization of the connection to the staging server by each instantiated task. This value is given by

$$T'_{f_0} = \frac{S}{BW} = \frac{9}{0.74} = 12.16\ Sec.$$

As can be seen from Figure 4.11, the rate of change in the number of bandwidth sharing links depends on the amount of variation in the simulation environment parameters, with a more gradual drop corresponding to increased variation in simulation parameters.

It can be also seen from Figure 4.11 that for tasks with a low complexity (corresponding to small values for T_c), total task throughput remains roughly unchanged as T_f increases up to T'_{f_0}. For the same range of values of T_f, the effective utilization of available computational servers remains at,or close to, its maximum value ($P_e \approx P$). As T_f exceeds T_{f_0}, computational servers utilization drops as total task turnaround time (data transfer

time + computational time) decreases due to absence of bandwidth sharing. This in turn leads to decrease in total task throughput as T_f further increases.

For tasks with high computational complexity (large values of T_c'), it can be seen that bandwidth sharing effects are not as dominant for $T_f' < T_{f_0}'$. Average task turnaround time is dominated by the computational time. Effective utilization of computational servers remains at its maximum value as T_f' increases. Total task throughput remains mainly dependent on the ability to keep all computational servers fully utilized.

Chapter 5

Conclusions and future work

In the preceding chapters, we presented research to study and implement a storage infrastructure that is integrated in the network fabric. This infrastructure allows for extending the resource sharing model of the network bandwidth to encompass storage resources. The integrated management of storage and bandwidth resources facilitates the optimization of certain classes of wide area distributed applications. In the following subsections, we outline some aspects of the IBP paradigm and its effects on distributed infrastructure, wide area applications, and content delivery mechanism.

5.1 Embedding storage into the network fabric

The design philosophy of IBP emphasizes the incorporation of user-level distributed storage resources into the fabric of the network itself. The ability to simultaneously manage network communication and storage resources provides an extra dimension to the parameter space that can be explored to enhance application functionality and/or performance. The IBP protocol itself follows a minimalist design approach that allows applications maximum flexibility in exploiting the combined communication/storage space made available through IBP.

The IBP protocol and storage characteristics add the dimensions of storage *durability* and *reliability* to the parameter space available for use by applications built on top of IBP. Through the judicious use of permanent, time-limited, stable, and volatile storage characteristics, applications have great flexibility in devising data replication and staging mechanisms that balance functionality, performance, and resource utilization. The use of these storage capabilities in a distributed collection of IBP depots, combined with information on network connectivity and bandwidth utilization, provide a powerful infrastructure for a plethora of distributed applications.

The IBP protocol provides a common underlying infrastructure for appli-

cation level routing. The flexibility obtained through the use of application-level routing must be balanced against the need to explicitly account for the different failure modes of a network transfer involving IBP depots. Deploying application-level routing at the coarse granularity level of entire files (or large file segments) ameliorates the scalability problems inherent in the small granularity, packet-level routing used in traditional Internet traffic.

5.2 IBP storage provisioning

The design of the IBP server provides storage resource owners with the ability to contribute owned storage resource as a part of a *IBP storage cloud* or a community of IBP storage depots. Such agglomeration of storage resources allows members of such a community (and others) to deploy application-level routing techniques as part of their distributed applications. Users get access to additional storage resources that are *distributed* across the network. Such IBP clouds can be built *on the fly* to support transient distributed application needs, or can be built on a more permanent basis as part of the overall network infrastructure.

The design of the IBP protocols and infrastructure simplifies the process of building distributed storage infrastructure that spans different adminis-

trative domains. In addition, deploying IBP storage within a single administrative domain can be an integral element in an array of computational resources. This can be seen, for example, in using IBP as *local* storage infrastructure in support of computational grids. Such storage can be used for staging input and output data associated with computational jobs. It can also be used to store intermediate results that may be needed at a later stage for subsequent computational steps.

5.3 IBP and logistical networking

The use of IBP to facilitate wide-area data transfer introduces a temporal dimension to the data routing problem. Traditional packet routing philosophy strives to deliver data to its destination *as soon as possible* while accounting for various network contingencies (congestion ..etc.). The use of IBP as part of a larger networking infrastructure allows for the development of different types of data routing algorithms, where it is possible to temporarily store in-transit data at an IBP depot pending availability of suitable network connection to its next logical destination. Used as such, IBP servers can be thought as large (but usually slower) router-attached back-end storage. Deploying router-attached IBP servers introduces another degree of freedom

in the data routing problem space, where a datum in transit can be forward to its next logical destination, dropped (and subsequently re-transmitted) due to network congestion or lack of buffer space, or off-loaded to attached IBP storage pending more opportune networking conditions. While it is not practical to actually use IBP at the packet routing level of traditional network traffic, using IBP at a coarser granularity can serve as the underlying layer for the development of general application level data routing techniques that can encompass various Quality of Service (QOS) aspects.

The development of the Network Storage Stack [14,57] has greatly facilitated the deployment of many applications that utilize logistical networking techniques. This stack, shown in Figure 5.1, uses IBP as the foundation upon which a general logistical networking infrastructure is built. The L-Bone (Logistical Backbone) is a distributed runtime layer that allows clients to perform IBP depot discovery. The exNode is a data structure for aggregation, analogous to the Unix inode. 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 that may be used like a file. The logistical tools and the logistical file system provide the Logistical Runtime System (LoRS) that provide applications with a high level API for the manipulation and movement of files accessed through exNodes, and stored on

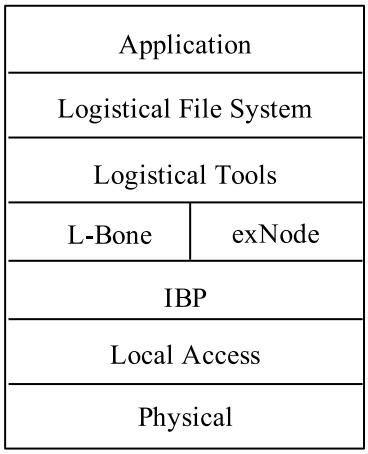


Figure 5.1: Logistical storage stack

IBP depots.

Various applications have been written to exploit the facilities provided by the logistical Storage Stack. Video IBPster [10] is video streaming tool, built using the tools of the Network Storage Stack, that delivers DVD-quality video without dropping frames, without losing data and without specialized multi-media streaming servers. In [34], IBP and logistical networking principles are used to developed a remote visualization system based on concepts of light field rendering, an Image Based Rendering (IBR) method using a 4-D plenoptic function. The system extends existing work on light fields by employing a modified method of parameterization and data or-

ganization that supports more efficient prefetching, caching and loss-less compression. Using this approach, a multi-gigabyte, high-resolution light field databases has been interactively browsed across the wide area network at 30 frames per second. In [52], the use of logistical networking and IBP to develop a scalable service for wide area computation, or programmable networking is explored. The resulting system , logistical networking computing, supports limited computation at intermediate nodes, while maintaining scalability of the overall system utilizing bandwidth, storage, and computational resources.

5.4 Future work

The IBP protocol and infrastructure represents a lower level layer upon which application level services are built. In this section we outline some of the directions into which research based on the IBP infrastructure can proceed. Such research can be categorized into two main areas, extending and enhancing the IBP layer itself, and building higher level abstractions on top of IBP.

5.4.1 Extending the IBP infrastructure

For wide adoption and deployment of the IBP infrastructure, the issues of data integrity, security and authenticity need to be addressed. The prototype implementation of IBP relies on the TCP/IP protocol for data transfer, guaranteeing data integrity *in transit*. However, should data get corrupted while in storage at an IBP storage depot, this event will go undetected by the IBP server code. The IBP protocol itself and/or the IBP server code can be extended to add measures to manage data integrity while in storage and/or in transit between IBP depots (e.g., by incorporating MD5 signatures into the IBP protocol). This requirement would be even more important should the IBP implementation use an unreliable communication protocol (e.g. UDP) for better performance.

Another possible extension area is the unlimited access to IBP storage in the current protocol and implementation. Access control is accomplished through th use of access capabilities that are assigned at allocation time. However, the storage allocation operation itself is not controlled. It is possible that IBP communities could aim to limit access to communal IBP storage to members of the community (and other approved entities). This can now be accomplished by controlling access to IBP communication ports us-

ing networking firewall techniques. The IBP protocol itself can be extended to handle this functionality. Such extension would necessarily increase the complexity of the IBP protocol and may introduce dependencies on other infrastructural technologies (e.g. Kerberos [55]).

5.4.2 Building services and abstractions on top of IBP

The true utility of IBP can only be realized as higher level abstractions and services are built on top of the IBP protocol and distributed storage infrastructure. It should be noted that the flexibility of directly manipulating IBP storage and network bandwidth is balanced by the complexity of devising such schemes that perform well when used in conjunction with other such schemes with different requirements and restrictions. It is important that an appropriate set of higher level abstractions and services be constructed on top of IBP that provide a simpler and more usable paradigm to user applications.

One of the more useful services that can be built on top of IBP is a generalized end-to-end data transfer service that allow clients to specify requirements and/or constraints on such transfer. Such specification would then be translated into a schedule of IBP data movements that strive to meet the specified requirements and constraints. The use of IBP to transport email

attachment is one such transfer where requirements and constraints are implicit in the problem definition. A more general service that allow clients to specify data transfer goals and constraints would extend the utility of IBP into more application domains.

The co-scheduling of storage and networking resources lies at the core of efficient use of IBP infrastructure. The ability to allocate different classes of IBP storage areas combine with the emerging deployment of networking Quality Of Service (QOS) mechanisms and protocols (e.g. RSVP) to provide a powerful and complex paradigm to provision data transfer and staging services. Such services would have QOS characterization that combines storage related metrics (e.g. reliability) with networking specific QOS measures to provide various levels of access to *storage in the network*. The mapping of user-level requirements into storage and networking policies and schedules is a challenging problem, especially in view of the dynamic nature of both resources.

It should be noted that recent developments and enhancements of IBP and logistical networking have already greatly extended the IBP infrastructure. Many of the ideas discussed here have already been implemented in different parts of the Network Storage Stack. Logistical networking is increasingly recognized as a viable and robust approach towards the efficient

utilization of the distributed storage, networking, and computing resources available to distributed applications.

Bibliography

Bibliography

[1] http://www.web-caching.com/research-projects.html, 2004.

[2] http://www.akamai.com/, 2004.

[3] D. Abramson, R. Sosic, J Giddy, and B Hall. Nimrod: A tool for performing parametised simulations using distributed workstations. In *The 4th IEEE Symposium on High Performance Distributed Computing*, August 1995.

[4] 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,*. http://dsl.cs.technion.ac.il/Starfish, 1999.

[5] Adnan Agbaria and James Plank. Design, implementation, and performance of checkpointing in NetSolve. In *International Conference on*

Dependable Systemsand Networks, FTCS-30 & DCCA-8, June 2000.

[6] A. Alexandrov, M. Ibel, K. E. Schauser, and C. Scheiman. Extending the operating system at the user level: the ufo global file system. In *USENIX 1997 Annual Technical Conference*, Januray 1997.

[7] W. Allcock, 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. *Journal of Network and Computer Applications*, 23:187–200, 2001.

[8] Dorian C. Arnold, Dieter Bachmann, and Jack Dongarra. Request sequencing: Optimizing communication for the grid. In *Euro-Par 2000*, 2000.

[9] S. Atchley, M. Beck, H. Hagewood, J. Millar, T. Moore, J. S. Plank, and S. Soltesz. Next generation content distribution using the logistical networking testbed. Technical report, University of Tennessee, 2002.

[10] S. Atchley, S. Soltesz, J. S. Plank, and M. Beck. Video IBPster. *Future Generation Computer Systems*, 19:861–870, 2003.

[11] Scott Atchley, Stephen Soltesz, Jamse S. Plank, and Micah Beck. Video IBPster. Technical report, University of Tennessee, 2002.

[12] S. Baru. Storage Resource Broker (SRB) reference manual. `http://www.npaci.edu/DICE/SRB/SRB10/index.html`, 1997.

[13] J. Basney, R. Raman, and M. Livny. High throughput Monte Carlo. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[14] A. Bassi, M. Beck, T. Moore, and J. S. Plank. The logistical backbone: Scalable infrastructure for global data grids. In *Asian Computing Science Conference2002*, Hanoi, Vietnam, December 2002.

[15] 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.

[16] M. Beck, T. Moore, and J. S. Plank. An End-to-End approach to globally scalable network storage. In *Proceedings of ACM Sigcomm 2002*. Association for Computing Machinery, 2002.

[17] M. Beck, T. Moore, J. S. Plank, and M. Swany. Logistical networking: Sharing more than the wires. In C. A. Lee S. Hariri and C. S. Raghavendra, editors, *Active Middleware Services*. Kluwer Academic, Norwell, MA, 2000.

[18] Micah Beck, Ying Ding, Erika Fuentes, and Sharmila Kancherla. An exposed approach to reliable multicast in heterogeneous logistical networks. In *Workshop on Grids and Advanced Networks*, May 2003.

[19] 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.

[20] S. Bhattacharjee, K. Calvert, and E. Zegura. Active networking and the end-to-end argument. In *Proceedings of ICNP '97*, October 1997.

[21] Andrew Birrell, Roy Levin, Roger M. Needham, and Michaael Schroeder. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[22] N. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies. `http://www.ietf.org/rfc/rfc1521.txt`, September 1993.

[23] C. M. Bowman, P. B. Danzig, D. R. Hardy, and U. Manbar. The Harvest information and discovery system. *Computer Networks and ISDN*

System, 28:119–125, 1995.

[24] 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*, August 1998.

[25] Brent Callaghan. WebNFS: The filesystem for the internet. `http://www.sun.com/software/white-papers/wp-webnfs/index.html`, April 1997.

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

[27] H. Casanova, M. Kim, J. S. Plank, and J. Dongarra. Adaptive scheduling for task farming with grid middleware. *Int. Jrnl. of High Perf. Comp.*, 13(3):231–240, Fall 1999.

[28] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS parameter sweep template: User-level middleware for the grid. In *SC00 Conference on High-Performance Computing*, November 2000.

[29] Henry Casanova and Jack Dongara. NetSolve: A Network-Enabled Server for Solving Computational Science Problems. *Int. Journal of Supercomp. Applic. and High Perf. Comp.*, 11(3), Fall 1997.

[30] V. Cerf and R. Kahn. A Protocol for packet network communication. *IEEE Trans. Communications*, Com-22(5):637–648, May 1974.

[31] Y. Chae, S. Merugu, E. Zegura, and S. Bhattacharjee. Exposing the network: Support for topology sensitive applications. In *Proceedings of IEEE OpenArch 2000*, March 2000.

[32] 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*, October 1999.

[33] David D. Clark. The design philosophies of the DARPA internet protocols. In *SIGCOMM Symposium proceedings on Communications architectures and protocols*, August 1988.

[34] Jin Ding, Jian Huang, Micah Beck, Shaotao Liu, Terry Moore, and Stephen Soltesz. Remote visualization by browsing image based

databases. In *SuperComputing*, Phoenix, AZ, USA, November 2003. Elsevier.

[35] 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.

[36] W. R. Elwasif, J. S. Plank, and M. Beck. IBP - internet backplane protocol: Infrastructure for distributed storage (v 0.2). Technical report, University of Tennessee, February 1999.

[37] M. Ferris, M. Mesnier, and J. More. NEOS and Condor: Solving optimization problems over the internet. Technical Report ANL/MCS-P708-0398, Argonne National Laboratory, March 1998. `http://www-fp.mcs.anl.gov/otc/Guide/TechReports/index.html`.

[38] R. Fielding, J. Gettys, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. `http://www.ietf.org/rfc/rfc2068.txt`, January 1997.

[39] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[40] Ian Foster and Carl Kesselman, editors. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, November 1998.

[41] 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.

[42] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. `http://www.ietf.org/rfc/rfc2045.txt`, November 1996.

[43] N. Freed and K. Moore. Definition of the URL MIME External-Body Access-Type. `http://www.ietf.org/rfc/rfc2017.txt`, October 1996.

[44] The Globus project. `http://www.globus.org/`.

[45] A. S. Grimshaw, W. A. Wulf, and The Legion Team. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[46] J. Reynolds J. Postel. File Transfer Protocol – FTP. `http://www.ietf.org/rfc/rfc959.txt`, October 1985.

[47] Brian Kantor and Phil Lapsley. Network News Transfer Protocol. `http://www.ietf.org/rfc/rfc0977.txt`, February 1986.

[48] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[49] U. Legedza, D. Wetherall, and J. Guttag. Improving the performance of distributed applications using active networks, 1998.

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

[51] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for unix. *ACM Trans. on Computer Systems*, 2(3):181–197, August 1984.

[52] Terence Moore, Micah Beck, and James S. Plank. An end-to-end approach to globally scalable programmable networking. In *Workshop*

on Future Directions in Network Architecture (FDNA'03), Karlsruhe, Germany, August 2003.

[53] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3), March 1986.

[54] H. Nakada, H. Takagi, S. Matsuoka, U. Nagashima, M. Sato, and S. Sekiguchi. Utilizing the metaserver architecture in the Ninf global computing system. In *High Perf. Comp. and Network. '98, LNCS 1401*, pages 607–616, 1998.

[55] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE communications*, 32(9), September 1994.

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

[57] J. S. Plank, S. Atchley, Y. Ding, and M. Beck. Algorithms for high performance, wide-area distributed file downloads. *Parallel Processing Letters*, 13(2):207–224, June 2003.

[58] 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.

[59] James S. Plank, Micah Beck, Wael R. Elwasif, Terence Moore, Martin Swany, and Rich Wolski. The internet backplane protocol: Storage in the network. In *Netstore 99: Network Storage Symposium*, October 1999.

[60] The Porcupine project. `http://porcupine.cs.washington.edu/`.

[61] Jon Postel and Joyce Reynolds. File Transfer Protocol (FTP). `http://www.ietf.org/rfc/rfc0959.txt`, October 1985.

[62] Jonathan B. Postel. Simple Mail Transfer Protocol – SMTP. `http://www.ietf.org/rfc/rfc0821.txt`, August 1982.

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

[64] Herman C. Rao and Larry L. Peterson. Accessing files in an internet: the Jade file system. *IEEE Trans. on Software Engineering*, 19(6):613–624, June 1993.

[65] P. Resnick and A. Walker. The text/enriched MIME Content-type. `http://www.ietf.org/rfc/rfc1896.txt`, February 1996.

[66] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany*, pages 329–350, November 2001.

[67] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles (SOSP'01) Lake Louise, Alberta, Canada*, October 2001.

[68] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[69] Yasushi Saito, Eric Hoffman, Brian Bershad, Henry Levy, and Bertil Foilot. The Porcupine Mail Server. *Eighth ACM Sigops European*

Workshop, September 1998.

[70] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. The design and implementation of the sun network file system. In *Proceedings of the 1985 USENIX Summer Conference*, pages 119–130, June 1985.

[71] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Seigel, and David C. Steere. Coda: A highly available file system for a distributed workstaion environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[72] Michaael Schroeder, Andrew D. Birrell, and Roger M. Needham. Experience with Grapevine: The growth of a distributed system. *ACM Trans. Comp. Sys.*, 2(1):3–23, February 1984.

[73] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima. Ninf: Network based information library for globally high performance computing. In *Proc. Parallel Object-Oriented Methods and Appl.*, pages 39–48, Santa Fe, 1996.

[74] L. M. Silva, B. Veer, and J. G. Silva. How to get a fault-tolerant farm. In *World Transputer Congress*, pages 923–938, Aachen, Ger-

many, September 1993.

[75] Sonar - a network proximity estimation service. http://icl.cs.utk.edu/projects/sonar/.

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

[77] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, 1997.

[78] A. Vahdat, P. Eastham, and T Anderson. WebFS: A global cache coherent filesystem. Technical report, Department of Computer Science, UC Berkeley, 1996.

[79] D. Wessels. Squid internet object cache. http://squid.nlanr.net/.

[80] R. Wolski, J. Brevik, C. Krintz, G. Obertelli, N. Spring, and A. Su. Running EveryWare on the computational grid. In *SC99 Conference on High-performance Computing*, November 1999.

[81] Rich Wolski, Neil Spring, and Jim Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 1998.

[82] W. Yeong, T. Howes, and S. Kille. Lightweight directory access protocol. `http://www.ietf.org/rfc/rfc1777.txt`, March 1995.

[83] R. M. Young. Euler's constant. *Math Gazette*, 75(472):187–190, 1991.

[84] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

Vita

Wael Elwasif was born in Mansoura, Egypt where he attended elementary, middle, and high school. He obtained his B. Sc. in Electronics Engineering from the Faculty of Engineering, Mansoura University in 1989. He worked as instructor at the Higher Technological Institute, 10'th Ramadan City, Egypt and at the department of automatic control and computers, faculty of Engineering, Mansoura University, Egypt. He obtained his M. Sc. in Applied Mathematics from Florida Institute of Technology in 1995, and M. Sc. in Computer Science from the University of Tennessee, Knoxville in 1999. He obtained his PhD in Computer Science from the University of Tennessee, Knoxville in 2004.