In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, California, May 2006

An Architecture for Internet Data Transfer

Niraj Tolia[†], Michael Kaminsky[‡], David G. Andersen[†], and Swapnil Patil[†]
[†]*Carnegie Mellon University and* [‡]*Intel Research Pittsburgh*

Abstract

This paper presents the design and implementation of DOT, a flexible architecture for data transfer. This architecture separates content negotiation from the data transfer itself. Applications determine *what* data they need to send and then use a new *transfer service* to send it. This transfer service acts as a common interface between applications and the lower-level network layers, facilitating innovation both above and below. The transfer service frees developers from re-inventing transfer mechanisms in each new application. New transfer mechanisms, in turn, can be easily deployed without modifying existing applications.

We discuss the benefits that arise from separating data transfer into a service and the challenges this service must overcome. The paper then examines the implementation of DOT and its plugin framework for creating new data transfer mechanisms. A set of microbenchmarks shows that the DOT prototype performs well, and that the overhead it imposes is unnoticeable in the wide-area. End-to-end experiments using more complex configurations demonstrate DOT's ability to implement effective, new data delivery mechanisms underneath existing services. Finally, we evaluate a production mail server modified to use DOT using trace data gathered from a live email server. Converting the mail server required only 184 lines-of-code changes to the server, and the resulting system reduces the bandwidth needed to send email by up to 20%.

1 Introduction

Bulk data transfers represent more than 70% of Internet traffic [3]. As a result, many efforts have examined ways to improve the efficiency and speed of these transfers, but these efforts face a significant deployment barrier: Most applications do not distinguish between their control logic and their data transfer logic. For example, HTTP and SMTP both interleave their control commands (e.g., the HTTP header, SMTP's "`mail from:`", etc.) and their data transfers over the same TCP connection. Therefore, new innovations in bulk data transfer must be reimplemented for each application. Not surprisingly, the rate of adoption of innovative transfer mechanisms, particularly in existing systems, is slow.

Data transfer applications typically perform two different functions. The first is *content negotiation*, which is very application-specific. For example, a Web download involves transmitting the name of the object, negotiating the language for the document, establishing a common format for images, and storing and sending cookies. The second function is *data transfer*, in which the actual data bits are exchanged. The process of data transfer is generally independent of the application, but applications and protocols almost always bundle these functions together.

Historically, data transfers have been tightly linked with content negotiation for several reasons. The first is likely expediency: TCP and the socket API provide a mechanism that is "good enough" for application developers who wish to focus on the other, innovative parts of their programs. The second reason is the challenge of naming. In order to transfer a data object, an application must be able to name it. The different ways that applications define their namespaces and map names to objects is one of the key differences between many protocols. For example, FTP and HTTP both define object naming conventions, and may provide different names for the same objects. Other protocols such as SMTP only name their objects implicitly during the data transfer.

As a concrete example of the cost of this coupling, consider the steps necessary to use BitTorrent [8] to accelerate the delivery of email attachments to mailing lists. Such an upgrade would require changes to each sender and receiver's SMTP servers, and modifications to the SMTP protocol itself. These changes, however, would only benefit email. To use the same techniques to speed Web downloads and reduce the load at Web servers would again require modification of both the HTTP protocol and servers.

We propose cleanly separating data transfer from applications. Applications still perform content negotiation using application-specific protocols, but they use a *transfer service* to perform bulk point-to-point data transfers. The applications pass the data object that they want to send to the transfer service. The transfer service is then responsible for ensuring that this object reaches the receiver. The simplest transfer service might accomplish this by sending the data via a TCP connection to the receiver. A more complex transfer service could implement the above BitTorrent data transfer techniques, making them available to SMTP, HTTP, and other applications.

Separating data transfer from the application provides several benefits. The first benefit is for application developers, who can re-use available transfer techniques instead

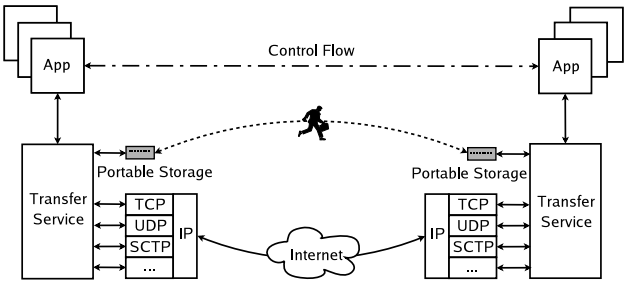


Figure 1: DOT Overview

of re-implementing them. The second benefit is for the inventors of innovative transfer techniques. Applications that use the transfer service can immediately begin using the new transfer techniques without modification. Innovative ideas do not need to be hacked into existing protocols using application-specific tricks. Because the transfer service sees the whole data object that the application wants to transfer, it can also apply techniques such as coding, multi-pass compression, and caching, that are beyond the reach of the underlying transport layers. The transfer service itself is not bound to using particular transports, or even established transports—it could just as well attempt to transfer the data using a different network connection or portable storage device.

Moving data transfer into a new service requires addressing three challenges. First, the service must provide a convenient and standard API for data transfer applications. Second, the architecture should allow easy development and deployment of new transfer mechanisms. Finally, the service must be able to support applications with diverse negotiation and naming conventions.

We present the design and implementation of a *Data-Oriented Transfer service*, or DOT, shown in Figure 1. The design of DOT centers around a clean interface to a modular, plugin based architecture to facilitate the adoption of new transfer mechanisms. DOT uses recent advances in content-based naming to name objects based upon their cryptographic hash, providing a uniform naming scheme across all applications.

This paper makes three contributions. First, we propose the idea of a data transfer service—a new way of structuring programs that do bulk data transfer, by separating their application-specific control logic from the generic function of data transfer. Second, we provide an effective design for such a service, its API and extension architecture, and its core transfer protocols. Finally, we evaluate our implementation of DOT with a number of micro- and macro-benchmarks, finding that it is easy to integrate with applications, and that by using DOT, applications can achieve significant bandwidth savings and easily take advantage of new network capabilities.

2 Transfer Service Scenarios

The advantage of a generic interface for data transfer is that it enables new transfer techniques across several applications. While we have implemented several transfer techniques within the DOT prototype, we believe its true power lies in the ability to accommodate a diverse set of scenarios beyond those in the initial prototype. This section examines several of these scenarios that we believe a transfer service enables, and it concludes with an examination of situations for which we believe the transfer service is inappropriate.

- A first benefit the transfer service could provide is cross-application caching. A DOT-based cache could benefit a user who receives the same file through an Instant Messaging application as well as via an email attachment. The benefits increase with multi-user sharing. An organization could maintain a single cache that handled all inbound data, regardless of which application or protocol requested it.

- Content delivery networks such as Akamai [1] could extend their reach beyond just the Web. A "data delivery network" could accelerate the delivery of Web, Email, NNTP, and any other data-intensive protocol, without customizing the service for each application. DOT could provide transparent access to Internet Backplane Protocol storage depots [4], to a storage infrastructure such as Open DHT [29], or to a collection of BitTorrent peers.

- The transfer service is not bound to a particular network, layer, or technology. It can use multi-path transfers to increase its performance. If future networks provided the capability to set up dedicated optically switched paths between hosts, the transfer service could use this facility to speed large transfers. The transfer need not even use the network: it could use portable storage to transfer data [14, 42].

- Finally, the benefits of the transfer service are not limited to simply exchanging bits. DOT creates the opportunity for making cross-application data processors that can interpose on all data transfers to and from a particular host. These proxies could provide services such as virus scanning or compression. Data processors combined with a delegation mechanism such as DOA [41] could also provide an architecturally clean way to perform many of the functions provided by today's network middleboxes.

The transfer service might be inappropriate for real-time communication such as `telnet` or teleconferencing. DOT's batch-based architecture would impose high latency upon such applications. Nor is the transfer service ideal for applications whose communication is primarily "control"

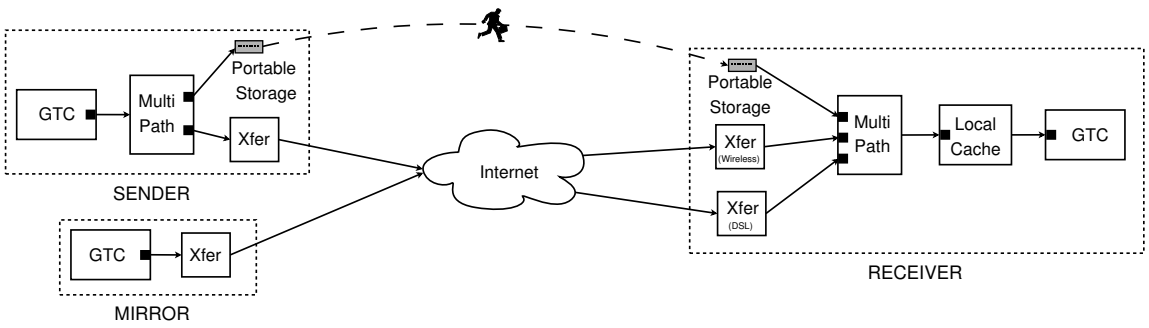


Figure 2: Example DOT Configuration. The GTC provides the transfer service functionality.

data, such as instant messaging with small messages. The overhead of the transfer service may wipe out any benefits it would provide. An important aspect of future work is to define this boundary more concretely—for instance, can the transfer service provide an interface to a multicast-based data transfer?

With these examples of the benefits that a transfer service could provide in mind, the next section examines a more concrete example of an example DOT configuration.

2.1 Example

Figure 2 shows an example DOT configuration that allows data transfers to proceed using multiple Internet connections and a portable storage device. This configuration provides three properties:

Surviving transient disconnection and mobility. Coping with disconnection requires that a transfer persist despite the loss of a transport-layer connection. In the example configuration, the multi-path plugin uses both the wireless and DSL links simultaneously to provide redundancy and load balancing. Mobility compounds the problem of disconnection because the IP addresses of the two endpoints can change. DOT's data-centric, content-based naming offers a solution to this problem because it is not tied to transport-layer identifiers such as IP addresses.

Transferring via portable storage. Portable storage offers a very high-latency, high-bandwidth transfer path [14, 42]. DOT's modular architecture provides an easy way to make use of such unconventional resources. A portable storage plugin might, for example, copy some or all of the object onto an attached disk. When the disk is plugged into the receiver's machine, the corresponding transfer plugin can pull any remaining data chunks from the disk. An advanced implementation might also make these chunks available to other machines in the network.

Using caching and multiple sources. The receiver can cache chunks from prior transfers, making them available to subsequent requests. The configuration above also shows how the transfer service can fetch data from multiple sources. Here, the multi-path plugin requests chunks in parallel from both the portable storage plugin and a set of network transfer plugins. The transfer plugins pull data from two different network sources (the sender and a mirror site) over two network interfaces (wireless and DSL).

3 Related Work

There are considerable bodies of work that have explored better ways to accomplish data transfers and architectures that insert a protocol between the application and the transport layers. We believe that DOT differs from prior work in choosing an architectural split (running as a service and primarily supporting point-to-point object transfers) that is both powerful enough to support a diverse set of underlying mechanisms, and generic enough to apply to a wide variety of applications.

Our design for DOT borrows from content-addressable systems such as BitTorrent [8] and DHTs. Like DHTs, DOT uses content-based naming to provide an application-independent handle on data.

DOT also bears similarity to the Internet Backplane Protocol (IBP) [4], which aims to unify storage and transfer, particularly in Grid applications. Unlike IBP, DOT does not specify a particular underlying method for data transfer; rather, DOT separates transfer methods from applications, so that future protocols like IBP could be implemented and deployed more rapidly.

At the protocol level, BEEP, the Blocks Extensible Exchange Protocol [32], is close in spirit to DOT. BEEP aims to save application designers from re-inventing an application protocol each time they create a new application, by providing features such as subchannel multiplexing and capability negotiation on top of underlying transport layers. BEEP is a protocol framework, available as a library against which applications can link and then extend to suit their own needs. BEEP's scope covers the application's content negotiation and data transfer. In contrast, DOT is a *service* that is shared by all applications; thus, a single new DOT plug-in can provide new transfer mechanisms or interpose on data to all applications.

Protocols such as FTP [26], GridFTP [35], ISO FTAM (ISO 8571), and even HTTP [15] can be used by appli-

cations to access data objects, either by invoking a client for that protocol or by implementing it within the application protocol. Many of the transfer techniques that distinguish these protocols (e.g., GridFTP's use of parallel data streams or negotiation of transfer buffer sizes) could be implemented as a DOT transfer plugin. By doing so, an unmodified "DOT-based" FTP client would then be able to take advantage of the new functionality, reducing the effort required to adopt the protocol enhancements.

Proxies are commonly used to process legacy application traffic in new ways. While DOT aims to be more general than application-specific examples such as Web proxies, it bears resemblance to generic proxies such as the DNS-based OCALA [18] or the packet capture approaches used by RON [2] and the X-bone [38] to re-route traffic to an overlay. The drawback of these more generic approaches is that they lack knowledge of what the application is attempting to do (e.g., transfer a certain block of data) and so become limited in the tools they can apply. However, some of the advantages of DOT can be realized through the use of protocol-specific proxies. For example, our modified email server can be used as a mail relay/proxy when co-located with unmodified mail servers.

The initial DOT plugins borrow techniques from several research efforts. Rhea et al. designed a Web proxy-to-proxy protocol that transfers content hashes to reduce bandwidth [30]. We show in Section 6.3 that DOT obtains similar benefits with email traffic. Spring and Wetherall use a similar hash-based approach to discover data duplication at the IP layer [34], and the `rsync` program uses a fingerprint-like approach to efficiently synchronize files across the network [39]. Mogul et al. showed similar benefits arising from using delta encoding in HTTP [22].

Finally, distributed object and file systems attempt to provide a common layer for implementing distributed systems. These systems range from AFS [17] and NFS [6] to research systems too numerous to cover in detail. Prominent among these are the storage systems that use content-addressable techniques for routing, abstracting identity, and saving bandwidth and storage [5, 9, 10, 12, 23, 27, 37]. Recent file systems have also incorporated portable storage for better performance [25, 36].

Like these distributed storage systems, DOT aims to mask the underlying mechanics of network data transfer from applications. Unlike these systems, DOT does not provide a single mechanism for performing the transfers. Its extensible architecture does not assume a "one size fits all" model for the ways applications retrieve their data. We do not wish to *force* the user of a DOT-based application to depend on an underlying distributed system if they only wish to perform point-to-point transfers. DOT complements many of these distributed systems by providing a single location where service developers can hook in, allowing applications to take advantage of their services.

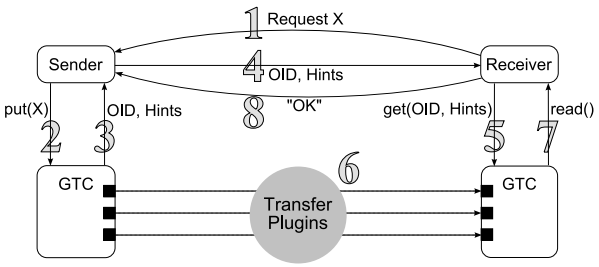


Figure 3: A data exchange using DOT

4 Design

This section first presents the basic system architecture, and then examines several details: (1) the manner in which applications use DOT; (2) the way the DOT transfer plugins and the default DOT transfer protocol operate; (3) the way that DOT accesses storage resources; and (4) the way that DOT plugins can be chained together to extend the system.

Applications interact with the transfer service as shown in Figure 1. Applications still manage their control channel, which handles content negotiation, but they offload bulk transfers to the transfer service. The transfer service delivers the data to the receiver using lower layer system and network resources, such as TCP or portable storage.

DOT is a receiver-pull, point-to-point service. We chose the receiver pull model to ensure that DOT only begins transferring data after both the sending and receiving applications are ready. We chose to focus on point-to-point service for two reasons: First, such applications represent *the* most important applications on the Internet (e.g., HTTP, email, etc.). Second, we believe that those point-to-multipoint applications that focus on bulk data transfer can be easily supported by DOT's plug-in model (e.g., transparent support for BitTorrent or CDNs).

Figure 3 shows the basic components involved in a DOT-enabled data transfer: the sender, receiver, and the DOT transfer service. The core of the transfer service is provided by the DOT Generic Transfer Client (GTC). The GTC uses a set of *plugins* to access the network and local machine resources:

- **Transfer plugins** accomplish the data transfer between hosts. Transfer plugins include the default GTC-to-GTC transfer plugin and the portable storage transfer plugin.
- **Storage plugins** provide DOT with access to local data, divide data into chunks, and compute the content hash of the data. Storage plugins include the disk-backed memory cache plugin used in our implementation, or a plugin that accesses locally indexed data on the disk.

Sending data using DOT involves communication

among the sender, receiver, and the GTC. Figure 3 enumerates the steps in this communication:

(1) The receiver initiates the data transfer, sending an application-level request to the sender for object X. Note that the sender could also have initiated the transfer.

(2) The sender contacts its local GTC and gives the GTC object X using the `put` operation.

(3) When the `put` is complete, the GTC returns a unique object identifier (OID) for X to the sender as well as a set of *hints*. These hints, described in Section 4.2.1, help the receiver know where it can obtain the data object.

(4) The sender passes the OID and the hints to the receiver over the application control channel.

(5) The receiver uses the `get` operation to instruct its GTC to fetch the object corresponding to the given OID.

(6) The receiver's GTC fetches the object using its transfer plugins, described in Section 4.2, and then

(7) returns the requested object to the receiver.

(8) After the transfer is complete, the receiver continues with its application protocol.

DOT names objects by OID; an OID contains a cryptographic hash of the object's data plus protocol information that identifies the version of DOT being used. The hash values in DOT include both the name of the hash algorithm and the hash value itself.

4.1 Transfer Service API

The application-to-GTC communication, shown in Table 1, is structured as an RPC-based interface that implements the **put** and **get** family of functions for senders and receivers respectively. A key design issue for this API is that it must support existing data-transfer applications while simultaneously enabling a new generation of applications designed from the ground-up to use a transfer service. Our experience implementing a DOT prototype has revealed several key design elements:

Minimal changes to control logic. Using the data transfer service should impose minimal changes to the control logic in existing applications. We created a stub library that provides a read/write socket-like interface, allowing legacy applications to read data from the GTC in the same way they previously read from their control socket. This interface requests that the GTC place data in-order before sending it to the application.

Permit optimization. The second challenge is to ensure that the API does not impose an impossible performance barrier. In particular, the API should not mandate extra data copies, and should allow an optimized GTC implementation to avoid unnecessarily hashing data to compute OIDs. Some applications also use special OS features (e.g., the zero-copy `sendfile` system call) to get high performance; the API should allow the use of such services or provide equivalent alternatives. To address such performance concerns, the GTC provides a file-descriptor passing API that sends data to the GTC via RPC.[1] DOT gives high-performance applications the choice to receive data out-of-order to reduce buffering and delays.

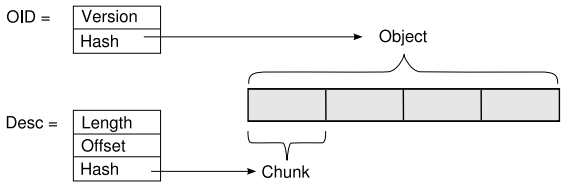


Figure 4: Relationship between DOT objects, chunks, OIDs and descriptors

Data and application fate sharing. The third design issue is how long the sender's GTC must retain access to the data object or a copy of it. The GTC will retain the data provided by the application at *least* until either: (a) the application calls `GTC_done()`; or (b) the application disconnects its RPC connection from the GTC (e.g., the application has exited or crashed). There is no limit to how long the GTC may cache data provided by the application—our implementation retains this data in cache until it is evicted by newer data.

4.2 DOT Transfer Plugins

The GTC transfers data through one or more transfer plugins. These plugins have a simple, standard interface that can implement diverse new transfer techniques. Internally, the GTC splits objects into a series of *chunks*. Each chunk is named with a *descriptor*, which contains a hash of the chunk, its length, and its offset into the object. Figure 4 shows the relationship between objects, OIDs, chunks, and descriptors. The transfer plugin API has three functions:

```
get_descriptors(oid, hints[], cb)
get_chunks(descs[], hints[], cb)
cancel_chunks(descriptors[])
```

A transfer plugin must support the first two functions. The `cb` parameter is an asynchronous callback function to which the plugin should pass the returned descriptors or chunks. Some plugins may not be able to cancel an in-progress request once it has gone over the network, and so may discard cancel requests if necessary.

To receive data, the GTC calls into a single transfer plugin with a list of the required chunks. That plugin can transfer data itself or it can invoke other plugins. Complex policies and structures, such as parallel or multi-path transfers, can be achieved by a cascade of transfer plugins that build upon each other. For example, the current DOT prototype provides a multi-path plugin which, when

[1] DOT does not yet cache object OIDs, but we note that systems such as EMC's Centera [13] already store the hash of files, so such an optimization is feasible.

Type	Command	Description
	PUT Commands	
t_id	GTC_put_init()	Initiates an object "put". Returns a transaction identifier.
void	GTC_put_data(t_id, data)	Adds object data to the GTC
(OID, Hints)	GTC_put_commit(t_id)	Marks the end of object data transfer. Returns an opaque structure consisting of the OID and Hints
(OID, Hints)	GTC_put_fd(file descriptor)	Optimized put operation that uses an open file descriptor
void	GTC_done(OID)	Allows the GTC to release resources associated with OID.
	GET Commands	
t_id	GTC_get_init(OID, mode, hints)	Initiates an object fetch. Returns a transaction identifier. Mode can be Sequential or Out-of-Order.
int	GTC_get(t_id, buf, &offset, &size)	Read data from the GTC. Returning zero indicates EOF.

Table 1: The Application-to-GTC API

instantiated, takes a list of other transfer plugins to which it delegates chunk requests.

Every GTC implementation must include a default GTC-GTC plugin that is available on all hosts. This plugin transfers data between two GTCs via a separate TCP connection, using an RPC-based protocol. The receiver's GTC requests a list of descriptors from the sender's GTC that correspond to the desired OID. Once it starts receiving descriptors, the receiver's GTC can begin sending pipelined requests to transfer the individual chunks.

4.2.1 Receiving data: Hints

As DOT is based on a receiver-pull model, hints are used to inform the receiver's GTC of possible data locations. They are generated by the sender's GTC, which the sender passes to the receiver over the application control channel. The receiver passes the hints to its GTC. Hints, associated with an OID, are only generic location specifiers and do not include any additional information on *how* the fetched data should be interpreted. Section 7 discusses how DOT could support mechanisms such as content encoding and encryption via additional per-chunk metadata.

A hint has three fields: *method*, *priority*, and *weight*. The method is a URI-like string that identifies a DOT plugin and then provides plugin-specific data. Some examples might be `gtc://sender.example.com:9999/` or `dht://OpenDHT/`. As with DNS SRV records [16], the priority field specifies the order in which to try different sources; the weight specifies the probability with which a receiver chooses different sources with the same priority. By convention, the sender's GTC will include at least one hint—itself—because the sender's GTC is guaranteed to have the data.

4.3 Storage Plugins

The main purpose of the GTC is to *transfer* data, but the GTC must sometimes store data locally. The sender's GTC must hold onto data from the application until the receiver is able to retrieve it. The receiver's GTC, upon retrieving data might need to reassemble out-of-order chunks before handing the data to the receiver, or may wish to cache the data to speed subsequent transfers.

The GTC supports multiple storage plugins, to provide users and developers with flexible storage options. Examples of potential back-ends to the storage plugins include in-memory data structures, disk files, or an SQL database. The current DOT prototype contains a single storage plugin that uses in-memory hash tables backed by disk. The storage plugin is asynchronous, calling back to the GTC once it stores the data.

All DOT storage plugins provide a uniform interface to the GTC. To add data to the storage plugin, the GTC uses an API that mirrors the application PUT API in Table 1.

In addition to whole object insertion, the storage plugins export an API for single chunk storage. This API allows the GTC or other plugins to directly cache or buffer chunk data. The lookup functions are nearly identical to those supported by the transfer plugins, except that they do not take a hints parameter:

```
put_chunk(descriptor, data)
release_chunk(descriptor)
get_descriptors(oid, cb)
get_chunks(descriptors[], cb)
```

4.4 Configuring Plugins

DOT transfer plugins are configured as a data pipeline, passing `get_descriptors` and `get_chunks` requests on to subsequent transfer plugins. A simple DOT configuration consists of the GTC, a local storage plugin,

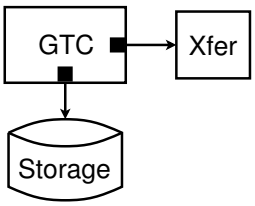


Figure 5: A simple DOT configuration

and a default GTC-GTC transfer plugin, shown in Figure 5. This configuration is instantiated as follows:

```
m = new gtcdMain();
sp = new storagePlugin(m);
xp = new xferPlugin(m);

m->set_xferPlugin(xp);
m->set_storagePlugin(sp);
```

Plugins that can push data, such as the portable storage plugin, need to be notified when new data becomes available. These plugins register with the GTC to receive notice when a new data object is inserted into the GTC for transfer, by calling `register_push()`.

5 Implementation

DOT is implemented in C++ using the libasync [21] library. Libasync provides a convenient callback-based model for creating event-driven services. DOT makes extensive use of libasync's RPC library, libarpc, for handling communication both on the local machine and over the network. For instance, the default GTC-GTC protocol is implemented using RPC.

5.1 Multi-path Plugin

The DOT multi-path plugin acts as a load balancer between multiple transfer plugins, each of which is bound to a particular network interface on a multi-homed machine. The plugin is configured with a list of transfer plugins to which it should send requests. The sub-plugins can be configured in one of two ways. Some plugins can receive requests in small batches, e.g., network plugins that synchronously request data from a remote machine. Other plugins instead receive a request for all chunks at once, and will return them opportunistically. The latter mode is useful for plugins such as the portable storage plugin that opportunistically discover available chunks.

The multi-path plugin balances load among its sub-plugins in three ways. First, it parcels requests to sub-plugins so that each always has ten requests outstanding. Second, to deal with slow or failed links, it supports request borrowing where already-issued requests are shifted from the sub-plugin with the longest queue to one with an empty queue. Third, it cancels chunk requests as they are satisfied by other sources.

5.2 Portable Storage Plugin

On the sender side, the portable storage plugin (PSP) registers to receive notifications about new OIDs generated by the GTC. When a new OID is discovered, the PSP copies the blocks onto the storage device. The implementation is naive, but effective: Each chunk is stored as a separate file named by its hash, and there is no index.

On the receiver, the PSP acts as a transfer plugin accessed by the multi-path plugin that receives a request for all descriptors. It polls the portable storage device every 5 seconds to determine if new data is available.[2] If the device has changed, the PSP scans the list of descriptors stored on the flash device and compares them to its list of requested descriptors, returning any that are available.

5.3 Chunking

The storage plugin also divides input data into chunks. To do so, it calls into a chunker library. The chunker is a C++ class that is instantiated for each data object sent to the storage plugin. It provides a single method, `chunk_data`, that receives a read-only pointer to additional data. It returns a vector of offsets within that data where the storage plugin should insert a chunk boundary. Our current implementation supports two chunkers, one that divides data into fixed-length segments, and one that uses Rabin fingerprinting [28, 20] to select data-dependent chunk boundaries. Rabin fingerprinting involves computing a function over a fixed-size sliding window of data. When the value of the function is zero, the algorithm signals a boundary. The result is that the boundaries are determined by the value of the data; they are usually the same despite small insertions or deletions of data.

5.4 Stub Library

Most applications are not written to send data using RPC calls to a local transfer service. To make it easy to port existing applications to use DOT, we created a socket-like stub library that preserves the control semantics of most applications. The library provides five functions:

```
dot_init_get_data(oid+hints)
dot_init_put_data()
dot_read_fn(fd, *buf, timeout)
dot_write_fn(fd, *buf, timeout)
dot_fds(*read, *write, *time)
```

`get` and `put` return a forged "file descriptor" that is used by the `read` and `write` functions. The `read` and

[2] Polling, while less efficient than receiving notification from the OS, was deemed more portable.

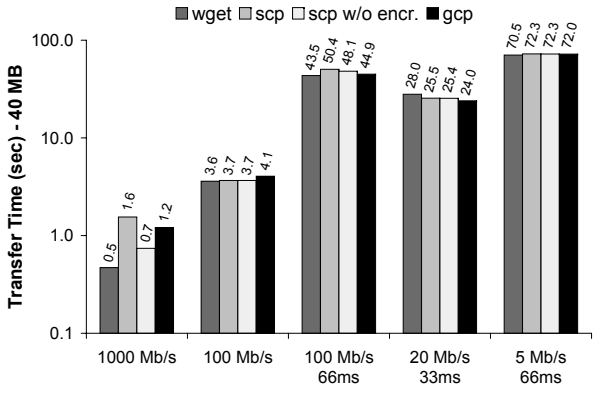


Figure 6: gcp vs. other standard file transfer tools

write functions provide a blocking read and write interface. The fds function allows select-based applications to query the library to determine which real file descriptors the application should wait for, and for how long. Section 6.3 shows how the stub library made it easy to integrate DOT with a production mail server package.

6 Evaluation

The primary goal of the DOT architecture is to facilitate innovation without impeding performance. This section first examines a number of microbenchmarks of basic DOT transfers to understand if there are bottlenecks in the current system, and to understand whether they are intrinsic or could be circumvented with further optimization. It then examines the performance of plugins that use portable storage and multiple network paths to examine the benefits that can arise from using a more flexible framework for data transfer. This section concludes by examining the integration of DOT into Postfix [40], a popular mail server.

6.1 Microbenchmarks

To demonstrate DOT's effectiveness in transferring bulk data, we wrote a simple file transfer application, gcp, that is similar to the secure copy (scp) program provided with the SSH suite. gcp, like scp, uses ssh to establish a remote connection and negotiate the control part of the transfer such as destination file path, file properties, etc. The bulk data transfer occurs via GTC-GTC communication.

We used this program to transfer files of sizes ranging from 4KB to 40MB under varying network conditions. In the interests of space, we present only the results from the 40MB file transfer as it highlights the potential overheads of DOT. These results, shown in Figure 6, compare the performance of gcp to wget, a popular utility for HTTP downloads, scp, and a modified version of scp that, like gcp, does not have the overhead of encryption for the bulk data transfer. All tools used the system's default values for

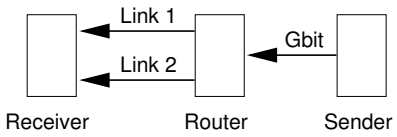


Figure 7: Topology for the multi-path evaluation. The capacities of links 1 and 2 are varied.

TCP's send and receive buffer size. gcp uses the fixed-size chunker for these experiments. All displayed results are the mean of 10 trials. All experiments were performed on a Gigabit Ethernet network at Emulab [43] with a dummynet [31] middlebox controlling network bandwidth and latency. The machines were Emulab's "pc3000" nodes with 3GHz 64-bit Intel® Xeon® processors and 2GB of DRAM.

For WAN conditions, gcp exhibits very little or no overhead when compared to the other file transfer tools and its performance is equivalent to scp both with and without encryption. On the local Gigabit network, gcp begins to show overhead. In this case, wget is the fastest. Unlike both scp and gcp, the Apache HTTP server is always running and wget does not pay the overhead of spawning a remote process to accept the data transfer. gcp is only slightly slower than scp. This overhead arises primarily because the GTC on the sender side must hash the data twice: once to compute the OID for the entire object and once for the descriptors that describe parts of the object. The hashing has two effects. First, the computation of the whole data hash must occur before any data can be sent to the receiver, stalling the network temporarily. Second, the hashes are relatively expensive to compute.

This overhead can be reduced, and the network stall eliminated, by caching OIDs (as noted earlier, some systems already provide this capability). While the computational overhead of computing the chunk hashes remains, it can be overlapped with communication. The computational overhead could also be reduced by generating a cheaper version of the OID, perhaps by hashing the descriptors for the object. However, the latter approach sacrifices the uniqueness of the OID and removes its usefulness as an end-to-end data validity check; we believe that retaining the whole-data OID semantics is worthwhile.

6.2 Plugin Effectiveness

This section examines the performance of the two transfer plugins we created for DOT, the multi-path plugin and the portable storage plugin.

6.2.1 Multi-Path Plugin

The multi-path plugin, described in Section 5.1, load balances between multiple GTC-GTC transfer plugins. We evaluate its performance using the same Emulab nodes as

Link 1	Link 2	single	multipath	savings
100/0	100/0	3.59	1.90	47.08%
	10/0	3.59	3.54	1.39%
100/33	100/33	21.46	11.15	48.04%
	10/33	21.46	13.58	36.72%
	1/33	21.46	20.44	4.75%
100/66	100/66	43.33	23.20	46.46%
	10/66	43.33	22.97	46.99%
	1/66	43.33	38.25	11.72%
10/66	10/66	48.39	23.42	51.60%
	1/66	48.39	39.20	18.99%
	0.1/66	48.39	44.14	8.78%
1/66	0.1/66	367.39	313.42	14.69%

Table 2: Multi-Path evaluation results. Links indicate bandwidth in Mbit/s and latency in milliseconds. The single column shows the time for a `gcp` transfer using only the fastest link of the pair.

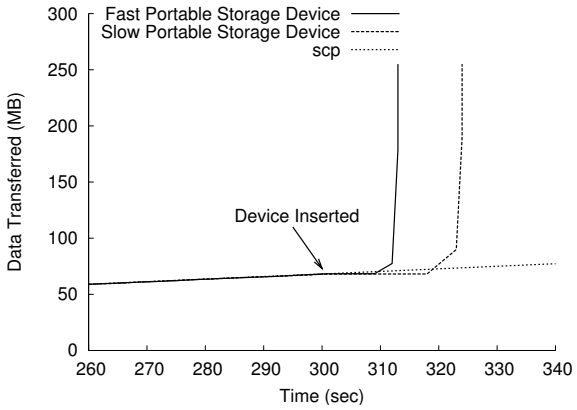


Figure 8: Portable Storage performance. The receiving machine completed the transfer from the USB flash device once it was inserted after 300 seconds. Without the flash device, the `scp` transfer took 1126 seconds to complete.

above. The receiver is configured with two network interfaces of lower capacity, and the sender with one Gigabit link, as shown in Figure 7.

Like the microbenchmarks, we examined the performance of the multi-path plugin in transferring 40MB, 4MB, and 400KB files via `gcp`. For brevity, we present only the 40MB results, but note that the performance on 400KB files was somewhat lower because they were not sufficiently large to allow TCP to consume the available capacity. All transfers were conducted using FreeBSD 5.4 with the TCP socket buffers increased to 128k and the initial slow-start flight size set to 4 packets.

Table 2 presents several combinations of link bandwidths and latencies, showing that the multi-path plugin can substantially decrease transfer time. For example, when load balancing between two directly connected 100 Mbit/s Ethernet links, the multi-path plugin reduced the transfer time from 3.59 seconds on a single link to 1.90 seconds. The best possible time to transfer a 40MB file over 100 Mbit/s Ethernet is 3.36 seconds, so this represents a substantial improvement over what a single link could accomplish. We note that the multi-path plugin was created and deployed with no modifications to `gcp` or the higher layer DOT functions.

In high bandwidth×delay networks, such as the 100 Mbit/s link with 66ms latency (representing a high-speed cross-country link), TCP does not saturate the link with a relatively small 40MB transfer. In this case, the benefits provided by the multi-path plugin are similar to those achieved by multiple stream support in GridFTP and other programs. Hence, using the multi-path plugin to bond a 100 Mbit/s and 10 Mbit/s link produces greater improvements than one might otherwise expect.

Finally, in cases with saturated links with high asymmetry between the link capacities (the second to last line in the table, a 10 Mbit/s link combined with a 100 Kbit/s link), the multi-path plugin provides some benefits by adding a second TCP stream, reducing the transfer time by roughly 9%. To understand the impact of request borrowing, we disabled it and re-ran the same experiment. Without request borrowing, the multi-path plugin slowed the transfer down by almost a factor of three, requiring 128 seconds to complete a transfer that the single link `gcp` completed in 48 seconds. Without request borrowing, the queue for the fast link empties, and the transfer is blocked at the speed of the slowest link until its outstanding requests complete.

6.2.2 Portable Storage

We evaluate DOT's use of portable storage using a workload drawn from *Internet Suspend/Resume (ISR)* [33]. ISR is a thick-client mechanism that allows a user to suspend work on one machine, travel to another location, and resume work on another machine there. The user-visible state at resume is exactly what it was at suspend. ISR is implemented by layering a virtual machine (VM) on a distributed storage system.

A key ISR challenge is in dealing with the large VM state, typically many tens of GB. When a user suspends a ISR machine, the size of *new* data generated during the session is in the order of hundreds of MBs. This includes both the changes made to the virtual disk as well as the serialized copy of the virtual memory image. Given the prevalence of asymmetric links in residential areas, the time taken to transfer this data to the server that holds VM state can be substantial. However, if a user is willing to carry a portable storage device, part of the VM state can be copied to the device at suspend time.

To evaluate the benefit of DOT in such scenarios, we simulate a fast residential cable Internet link with a maximum download and upload speed of 4 Mbit/s and 2 Mbit/s respectively. An analysis of data collected from the ISR

test deployment on Carnegie Mellon's campus [24] revealed that the average size of the data transferred to the server after compression is 255 MB. To model this scenario, we use DOT to transfer a single 255 MB file representing the combined checkin state. A USB key chain is inserted into the target machine[3] approximately five minutes after the transfer is initiated. The test uses two USB devices, a "fast" device (approx. 20MB/sec. read times) and a "slow" device (approx. 8MB/sec).

The speed of the transfer and total time for competition is presented in Figure 8. Immediately after the portable storage is inserted, the system experiences a small reduction in throughput as it scans the USB device for data.[4] As it starts reading the data from the device, the transfer rate increases substantially. Once the data has been read and cached, the transfer completes almost instantly.

6.3 Case Study: Postfix

To evaluate the ease with which DOT integrates with existing, real networked systems, we modified the Postfix mail program to use DOT, when possible, to send email messages between servers. Postfix is a popular, full-featured mail server that is in wide use on the Internet, and represented a realistic integration challenge for DOT.

We chose to examine the benefits of running DOT on a mail server for a number of reasons. First, mail is a borderline case for a mechanism designed to facilitate large data transfers. Unlike peer-to-peer or FTP downloads, most mail messages are small. Mail provides an extremely practical scenario: the servers are complex, the protocol has been around for years and was not designed with DOT in mind, and any benefits DOT provides would be beneficial to a wide array of users.

Postfix is a modular email server, with mail sending and receiving decomposed into a number of separate programs with relatively narrow functions. Outbound mail transmission is handled by the *smtp* client program, and mail receiving is handled by the *smtpd* server program. Postfix uses a "process per connection" architecture in which the receiving demon forks a new process to handle each inbound email message. These processes use a series of blocking calls, with timeouts, to perform network operations. Error handling is controlled via a setjmp/longjmp exception mechanism.

DOT was integrated into this architecture as an SMTP protocol extension. All DOT-enabled SMTP servers, in addition to the other supported features, reply to the EHLO greeting from the client with a "X-DOT-DATA" response. Any SMTP client compliant with the RFC [19] can safely ignore unrecognized SMTP options beginning with "X". This allows non-DOT enable clients to use the standard method of data transfer and allows the server to be backward compatible.

On the presentation of X-DOT-DATA by the server, any DOT-enabled client can use the X-DOT-DATA command as a replacement for the "DATA" command. Clients, instead of sending the data directly to the server, only send the OID and hints to the server as the body of the X-DOT-DATA command. Upon receipt of these, the server opens a connection to its local GTC and requests the specified data object. The server sends a positive completion reply only after successfully fetching the object. In the event of a GTC error, the server assumes that the error is temporary and a *transient* negative completion reply is sent to the client. This will make sure that the client either retries the DOT transfer or, after a certain number of retries, falls back to normal DATA transmission.

6.3.1 Mail Server Trace Analysis

The first results in this section are analytical results from a mail trace taken on a medium-volume research group mail server. This analysis serves two purposes. First, we examine the benefits of DOT's content-hash-based chunked encoding and caching. Chunked encoding is well known to benefit bulk peer-to-peer downloads [8] and Web transfers [30]; we wished to demonstrate that these benefits extend to an even wider range of applications. The second purpose is to generate a trace of email messages with which to evaluate our modified Postfix server.

Each message in the trace records an actual SMTP conversation with the mail server, recorded by a Sendmail mail filter. The traces are anonymized, reporting the keyed hash (HMAC) of the sender, receiver, headers, and message body. The anonymization also chunks the message body using a static sized chunk and Rabin fingerprints and records the hashes of each chunk, corresponding to the ways in which DOT could chunk the message for transmission. The subsequent analysis examines how many of these chunks DOT would transmit, but does *not* include the overhead of DOT's protocol messages. These overheads are approximately 100 bytes per 20KB of data, or 0.5%, and are much smaller than the bandwidth savings or the variance between days.

The email workload was an academic research group, with no notable email features (such as large mailing lists). We hypothesize that the messages in this workload are somewhat smaller than they would be in a corporate environment with larger email attachments, but we lack sufficient data about these other environments. The mail server handles approximately 2,500 messages per day. The traces cover 458,861 messages over 159 days. The distribution of email sizes, shown in Figure 9, appears heavy-tailed and consistent with other findings about email distribution. The sharp drop at 10-20MB represents common cut-off values

[3]Both machines have 3.2GHz Intel® Pentium® 4 CPUs with 2GB of SDRAM and run the 2.6.10-1.770-SMP Linux kernel.

[4]This slowdown could be avoided by spawning a helper process to perform the disk I/O.

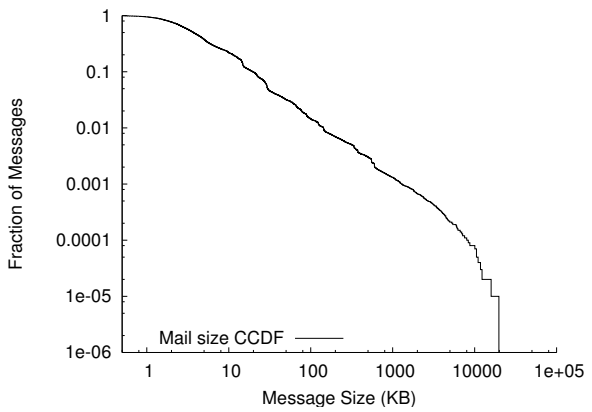


Figure 9: Mail message size distribution follows a heavy-tailed distribution until message size limits are reached.

for allowed message sizes. We eliminated one message that was 239MB, which we believe was a test message sent by the server administrator.

This section examines four different scenarios. **SMTP default** examines the number of bytes sent when sending entire, unmodified messages. With **DOT body**, the mail server sends the headers of the message directly, and uses DOT to transfer the message body. Only whole-file caching is performed: either the OID has been received, or not. With **Rabin body**, the mail server still sends the headers separately, but uses Rabin fingerprinting to chunk the DOT transfer. Finally, **Rabin whole** sends the entire message, headers included, using DOT. Because the headers change for each message, sending a statically-chunked OID for the entire message is unlikely to result in significant savings. The Rabin whole method avoids this problem, at the cost of some redundancy in the first content block. Allowing the GTC to chunk the entire message also allows the simplest integration with Postfix by avoiding the need for the mail program to parse the message content when sending. Our analysis assumes that DOT is enabled on both SMTP clients and servers. Table 3 shows the number of bytes sent by DOT in these scenarios.

In all, DOT saves approximately 20% of the total message bytes transferred by the mail server. These benefits arise in a few ways. As can be seen in Figure 10, a moderate number of messages are duplicated exactly once, and a small number of messages are duplicated many times—nearly 100 copies of one message arrived at the mail server. Second, as Table 3 showed, there is considerable partial redundancy between email messages that can be exploited by the Rabin chunking.

While our study did not include the number of large email attachments that we believe are more common in corporate environments, we did observe a few such examples. 1.5% of the bytes in our trace came from a 10 MByte email that was delivered eleven times to local users. The administrator of the machine revealed that a non-local

Method	Total Bytes	Percent Bytes
SMTP default	6800 MB	-
DOT body	5876 MB	86.41 %
Rabin body	5056 MB	74.35 %
Rabin whole	5496 MB	80.81 %

Table 3: Savings using different DOT integration methods. This table does not include DOT overhead bytes.

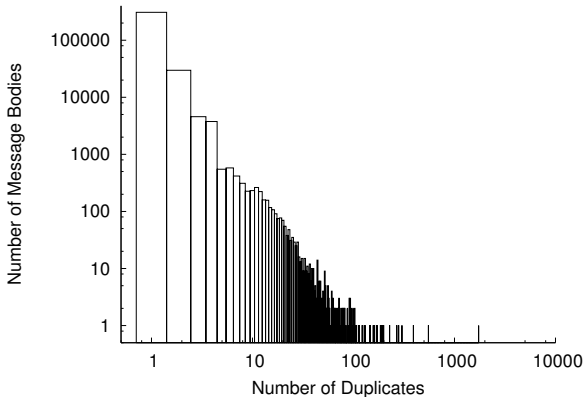


Figure 10: Histogram of the number of repetitions of particular message bodies. There were 307,969 messages with unique bodies, 29,746 with a body duplicated once, and so on. One message was duplicated 1,734 times.

mailing list to which several users were subscribed had received a message with a maximum-sized email attachment. Such incidents occur with moderate frequency, and are a common bane for email administrators. By alleviating the bandwidth, processing, and storage pain from such incidents, we hope that DOT can help allow users to communicate in the way *most convenient to them*, instead of following arbitrary decrees about application suitability.

6.3.2 System throughput

To evaluate DOT's overhead, we generated 10,000 email messages from the beginning of the mail trace. Each message is the same size as a message from the trace, and the message's content is generated from Rabin-fingerprint generated hash blocks in the trace message. Each unique hash value is deterministically assigned a unique content chunk by seeding a pseudo-random number generator with the hash value. The generated emails preserve some of the similarity between email messages, but because we cannot regenerate the original content (or other content that contains the same Rabin chunk boundaries), the generated emails are somewhat less redundant than the originals.

We replayed the 10,000 generated messages through Postfix running on the same machines used for the portable storage evaluation, connected with 100 Mbit/s Ethernet. Table 4 shows that the DOT-enabled Postfix required only

Program	Seconds	Bytes sent
Postfix	468	172 MB
Postfix-DOT	468	117 MB

Table 4: Postfix throughput for 10,000 email messages

Program	Original LoC	New LoC	%
GTC Library	-	421	-
Postfix	70,824	184	0.3%
smtpd	6,413	107	1.7%
smtp	3,378	71	2.1%

Table 5: Lines of Code Added or Modified in Postfix

68% of the total bandwidth, including all protocol overhead, but both systems had identical throughput.

The Postfix workload is extremely disk-seek bound. Neither the considerable bandwidth savings or the slight CPU overhead provided by DOT was sufficient to change this. We therefore believe that the widespread adoption of DOT in mail servers would therefore have a positive effect on bandwidth use without imposing noticeable overhead.

6.3.3 Integration Effort

Integrating DOT with Postfix took less than a week for a graduate student with no knowledge of Postfix. This time includes the time required to create the adapter library for C applications that do not use libasync, discussed in Section 5.4. Table 5 presents the number of lines of code (LoC) needed to enable DOT within Postfix. The modifications touched two Postfix subsystems, the `smtpd` mail server program, and the `smtp` mail client program. While we have only used two applications thus far with DOT (Postfix and `gcp`), we are encouraged by the ease with which DOT integrated with each.

7 Discussion

In previous sections, we presented the benefits we have observed using our initial DOT implementation, and discussed some of the benefits we believe DOT can realize from a transfer service in other scenarios. Our experience with DOT thus far has revealed several lessons and remaining challenges.

Efficiency. Our design of the DOT default transfer protocol assumes that its extra round trips can be amortized across file transfers of sufficient length. While this design is effective for DOT's intended target of large transfers, we would like to make the transfer service efficient for as many applications as possible, even if their transfers are small. For example, highly interactive Web services such as Google go to considerable effort to reduce the number of round-trips experienced by clients, and providers such as Akamai tout such reduction as a major benefit of their service. As the email study in Section 6.3 noted, email and many other protocols have a heavy tailed distribution with numerous small files being transferred.

There are two likely approaches that can reduce DOT's overhead for small transfers: either allow the application to send small objects directly, or implement a transfer plugin that passes small objects in-line as a hint over its control channel to bypass the need for additional communication. For reasons discussed below, we believe the former may be a better approach.

Application resilience. The DOT-enabled Postfix falls back to normal SMTP communication if either the remote host does not support DOT or if it cannot contact the GTC running on the local machine. This fallback to direct transfer makes the resulting Postfix robust enough to run as the primary mail program for one of our personal machines. As Cappos and Hartman noted on Planetlab, applications that depend on cutting-edge services are wise to fall back to simpler and more reliable mechanisms [7]. We believe this is sound advice for any application making use of DOT.

Security. In a system such as DOT, applications have several choices that trade privacy for efficiency. The simplest way for an application to ensure privacy is to encrypt its data before sending it to the transfer service. Unfortunately, this places restrictions on the transfer service's ability to cache data and obtain data from independent sources. Another simple (and common) choice is to not encrypt the data at all, and trust in whatever host or network security mechanisms are available. This approach allows the transfer service the most opportunities to exploit features of the data to improve transfers, but offers little security to the user. A promising middle ground is to support convergent encryption [11], in which data blocks are encrypted using their own hash values as keys. Only recipients that know the hashes of the desired data can decrypt the data blocks.

DOT must interact with many different applications, each of which may have its own mechanisms and policies for providing security and privacy. We feel strongly that a transfer service must support a wide variety of application requirements, and should not impose a particular security model or flavor of cryptography or key management. However, an efficient implementation of convergent encryption requires some support from the transfer service: The algorithm that splits data into chunks must be consistent across implementations, or the resulting encrypted blocks cannot be effectively cached. We intend to support convergent encryption in the near future.

Application preferences and negotiation. The current implementation of DOT performs data transfers completely independently of the application. While this suffices for a large and important class of applications, we believe that DOT can also benefit applications that desire more control over how their transfers are effected. Examples of such applications include those that have specific encryption requirements or that wish to take advantage of

different sending rates or quality of service categories.

Providing convergent encryption illustrates some of the problems that we must address. Applications that require encryption must be able to inform the transfer layer of their needs, and confirm that the data *will* be encrypted properly. Furthermore, the sender and receiver GTCs must also negotiate the availability of specific capabilities in case they are running different versions of DOT or have different plugins. Finally, the convergent encryption plugins must communicate the encryption keys.

The design of capability discovery and negotiation is an important part of our future work. As a first cut at supporting plugins that must negotiate specific capabilities, we are adding two types of metadata to DOT: per-object metadata that allows the receiving side to construct the plugin-chain required to process the data; and per-chunk metadata that allows the plugins to send information (such as encryption keys) needed to process the chunks when they are received.

Selecting Plugins and Data Sources. A DOT receiver may be able to obtain its desired data from multiple sources using several different plugins. The DOT architecture uses a hierarchy of plugins, some of which implement selection policies, to arbitrate between sources. If a variety of plugins become available, DOT's plugin configuration interface may need to evolve to support more sophisticated configurations that allow plugins to determine the capabilities of their upstream and downstream plugins. For example, it may be advantageous for a multi-path plugin to know more about the capacities of attached links or storage devices. While we believe such an interface could be useful, its design would be purely speculative in the absence of a wide variety of plugins from which to choose.

Several of our plugins represent promising starts more than finished products. We plan to enhance the multi-path plugin to support fetching data from mirror sites and multi-homed servers as well as multi-homed receivers. We are beginning the initial design for a "rendezvous" service that allows portable storage to be plugged in to a third party machine, instead of directly to the receiver.

Supporting dynamically generated objects. Our design calls for DOT to handle dynamically generated objects that cannot be fully hashed before transmission by assigning them a random OID. This change requires a small modification to the API to return the OID before the `put` call has completed, and requires that the remote `get_descriptors` call be able to return an indicator that the receiver should continue checking for more descriptors. A drawback is that random OIDs sever the tie between the OID and the data contents, which prevents per-object (but not per-chunk) caching.

Exploiting structure in data. A final issue in the design of DOT is the division of labor between the application and the transfer service in dividing data into chunks. In our design, the transfer service is solely responsible for chunking. As our evaluation of email traces showed, the use of Rabin fingerprinting can remove the need for application-specific chunking decisions in some cases, but there may remain other cases in which application knowledge is helpful. For instance, many applications transfer structured data. Databases, for example, may return data that has repetitions at the row level, which may be much smaller than DOT's default chunk size. While our current techniques work well for email and Web objects, we believe this issue merits further exploration.

8 Conclusion

This paper presented the design and implementation of an extensible data-oriented transfer service, DOT. DOT decouples application-specific content negotiation from the more general process of transferring data across the network. Using such a transfer service reduces re-implementation at the application layer and facilitates the adoption of new technologies to improve data transfer. Through a set of microbenchmarks and an examination of a production mail server modified to use DOT, we have shown that the DOT architecture imposes little overhead. DOT provides significant benefits, reducing bandwidth use and making new functionality such as multi-path or portable storage-based transfers readily available.

While our design still faces several challenges, we believe that introducing data transfer as a system *service* is a worthwhile goal. A widely deployed transfer service helps in the evolution of new services: researchers could easily try out new protocols using real, unmodified applications; and a significant fraction of Internet traffic could make use of new network-layer functions by simply adding a new transfer plugin. We believe that DOT could provide significant benefits to applications, networks, and ultimately, to users.

Acknowledgments

We are grateful to Hari Balakrishnan, Nick Feamster, Michael Freedman, our shepherd John Hartman, Larry Huston, Brad Karp, Dina Katabi, Mahim Mishra, M. Satyanarayanan, Alex Snoeren, Eno Thereska, and our anonymous reviewers for their valuable feedback. We thank Emulab for providing resources for evaluating DOT. This research was supported by NSF CAREER award CNS-0546551, by NSF grant CNS-0509004, by Intel Research, and by a grant from the Carnegie Mellon CyLab.

References

[1] Akamai. http://www.akamai.com, 1999.

[2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 131–145, Banff, Canada, Oct. 2001.

[3] N. B. Azzouna and F. Guillemin. Analysis of ADSL traffic on an ip backbone link. In *Proc. IEEE Conference on Global Communications (GlobeCom)*, San Francisco, CA, Dec. 2003.

[4] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable network storage. In *Proc. ACM SIGCOMM*, Pittsburgh, PA, Aug. 2002.

[5] T. C. Bressoud, M. Kozuch, C. Helfrich, and M. Satyanarayanan. OpenCAS: A flexible architecture for content addressable storage. In *2004 International Workshop on Scalable File Systems and Storage Technologies*, San Francisco, CA, Sept. 2004.

[6] B. Callaghan, B. Pawlowski, and P. Staubach. *NFS Version 3 Protocol Specification*, June 1995. RFC 1813.

[7] J. Cappos and J. Hartman. Why it is hard to build a long-running service on PlanetLab. In *Proc. Workshop on Real, Large Distributed Systems (WORLDS)*, San Francisco, CA, Dec. 2005.

[8] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003.

[9] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. 5th USENIX OSDI*, Boston, MA, Dec. 2002.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

[11] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proc. 22nd Intl. Conf on Distributed Computing Systems*, Vienna, Austria, July 2002. ISBN 0-7695-1585-1.

[12] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proc. HotOS VIII*, pages 75–80, Schloss-Elmau, Germany, May 2001.

[13] *EMC Centera Content Addressed Storage System*. EMC Corporation, 2003. http://www.emc.com/.

[14] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proc. ACM SIGCOMM*, pages 27–34, Karlsruhe, Germany, Aug. 2003.

[15] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol—HTTP/1.1*. Internet Engineering Task Force, Jan. 1997. RFC 2068.

[16] A. Gulbrandsen, P. Vixie, and L. Esibov. *A DNS RR for specifying the location of services (DNS SRV)*. Internet Engineering Task Force, Feb. 2000. RFC 2782.

[17] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6 (1), Feb. 1988.

[18] J. Kannan, A. Kubota, K. Lakshminarayanan, I. Stoica, and K. Wehrle. Supporting legacy applications over i3. Technical Report UCB/CSD-04-1342, University of California, Berkeley, May 2004.

[19] J. Klensin. *Simple Mail Transfer Protocol*. Internet Engineering Task Force, Apr. 2001. RFC 2821.

[20] U. Manber. Finding similar files in a large file system. In *Proc. Winter USENIX Conference*, pages 1–10, San Francisco, CA, Jan. 1994.

[21] D. Mazières, F. Dabek, E. Peterson, and T. M. Gil. Using libasync. http://pdos.csail.mit.edu/6.824-2004/doc/libasync.ps.

[22] J. C. Mogul, Y. M. Chan, and T. Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proc. First Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.

[23] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

[24] P. Nath, M. Kozuch, D. O'Hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, and M. Toups. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2006. (To appear).

[25] E. B. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the blue file system. In *Proc. 6th USENIX OSDI*, pages 363–378, San Francisco, CA, Dec. 2004.

[26] J. B. Postel and J. Reynolds. *File Transfer Protocol (FTP)*. Internet Engineering Task Force, Oct. 1985. RFC 959.

[27] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, CA, Jan. 2002.

[28] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[29] S. C. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proc. ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.

[30] S. C. Rhea, K. Liang, and E. Brewer. Value-based web caching. In *Proc. Twelfth International World Wide Web Conference*, Budapest, Hungary, May 2003.

[31] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communications Review*, 27(1): 31–41, 1997. ISSN 0146-4833.

[32] M. Rose. *On the Design of Application Protocols*. Internet Engineering Task Force, Nov. 2001. RFC 3117.

[33] M. Satyanaranyanan, M. A. Kozuch, C. J. Helfrich, and D. R. O'Hallaron. Towards seamless mobility on pervasive hardware. *Pervasive and Mobile Computing*, 1(2):157–189, 2005.

[34] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proc. ACM SIGCOMM*, Stockholm, Sweden, Sept. 2000.

[35] The Globus Project. GridFTP: Universal data transfer for the Grid. http://www-fp.globus.org/datagrid/deliverables/C2WPdraft3.pdf, Sept. 2000.

[36] N. Tolia, J. Harkes, M. Kozuch, and M. Satyanarayanan. Integrating portable and distributed storage. In *Proc. 3rd USENIX Conference on File and Storage Technologies*, San Francisco, CA, Mar. 2004.

[37] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, A. Perrig, and T. Bressoud. Opportunistic use of content addressable storage for distributed file systems. In *Proc. USENIX Annual Technical Conference*, pages 127–140, San Antonio, TX, June 2003.

[38] J. Touch and S. Hotz. The X-Bone. In *Proc. 3rd Global Internet Mini-Conference in conjunction with IEEE Globecom*, pages 75–83, Sydney, Australia, Nov. 1998.

[39] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, The Australian National University, Canberra, Australia, 1996.

[40] W. Venema. The Postfix home page. http://www.postfix.org/.

[41] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *Proc. 6th USENIX OSDI*, San Francisco, CA, Dec. 2004.

[42] R. Y. Wang, S. Sobti, N. Garg, E. Ziskind, J. Lai, and A. Krishnamurthy. Turning the postal system into a generic digital communication mechanism. In *Proc. ACM SIGCOMM*, pages 159–166, Portland, OR, Aug. 2004.

[43] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th USENIX OSDI*, pages 255–270, Boston, MA, Dec. 2002.