

What the Protocol Stack Missed: The Transfer Service

Niraj Tolia
Carnegie Mellon University

David G. Andersen
Carnegie Mellon University

Michael Kaminsky
Intel Research Pittsburgh

Swapnil V. Patil
Carnegie Mellon University

ABSTRACT

This WIP proposes a new architecture for applications that perform bulk data transfers. This architecture, called DOT (for *data-oriented transfer*), cleanly separates out two functions that are commingled in today's applications. Using DOT, applications perform content *negotiation* to determine what content to send. They then pass that data object to the transfer service to perform the actual data transmission. This separation increases application flexibility, enables the rapid development of innovative transfer mechanisms, reduces developer effort, and allows increased efficiency through cross-application sharing.

1. INTRODUCTION

Bulk data transfers represent more than 70% of Internet traffic. As a result, many efforts have examined ways to improve the efficiency and speed of these transfers. Unfortunately, these innovative transfer mechanisms face a serious barrier to implementation and widespread deployment because they must be reimplemented for each application. This barrier arises because most applications do not distinguish between their control logic and their data transfer logic.

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

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 to accelerate email attachment delivery to mailing lists. Such an upgrade would require changes to each sender and receiver's SMTP servers, and modifications to the SMTP protocol itself. However, to use the same techniques to speed web downloads would again require modification of both the HTTP protocol and servers.

Decoupling data transfer from the application provides several benefits. The first benefit is for application developers, who can reuse available transfer techniques instead 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.

This work provides a blueprint for a *data-oriented transfer service*, or DOT. DOT's design centers around a clean interface to a modular, plugin based architecture to facilitate the adoption of new transfer mechanisms. Using recent advances in content-based naming, DOT provides a uniform naming scheme across all applications. The contributions of DOT are twofold. 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.

2. ARCHITECTURE OVERVIEW

Applications interact with the transfer service as shown in Fig-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

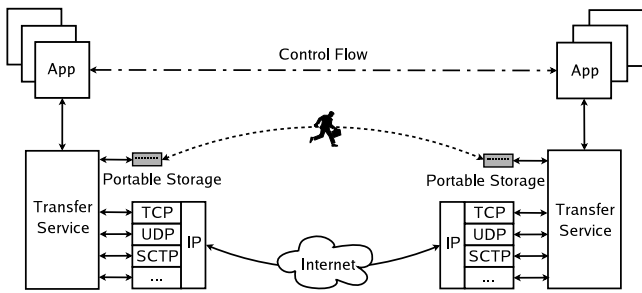


Figure 1: DOT Overview

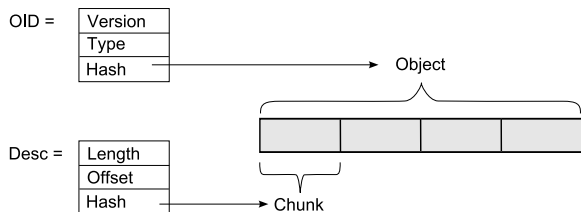


Figure 2: Relationship between DOT objects, chunks, OIDs and descriptors

ure 1. Applications still manage the control channel, which handles content negotiation, but they offload bulk data 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.

Using DOT, the sending application gives its data to the transfer service, which returns an object ID (OID). The application transmits this OID via the control channel. The receiving application in turn requests this OID from its own transfer service, which coordinates with the sender’s to transmit the desired data. Along with the OID, the application transmits a set of “hints” that tell the receiver’s transfer service where it might look to find the data (e.g., directly from the sender, from a third-party mirror site, and so on).

DOT provides a default transfer mechanism by which receivers request data directly from the sender. This mechanism is a block-based RPC protocol. The sender GTC divides the object up into a series of chunks, either statically or by using Rabin fingerprints. Using the default protocol, the receiver first requests a list of what chunks belong to a particular OID, and then requests each of these chunks in series. Using this division permits DOT to effectively cache partially-completed transfers and eliminate redundant transmissions.

Using the data transfer service imposes minimal changes to the control logic common in existing applications. We have created a simple library that provides a read/write socket-like interface; using this library, we ported the popular *postfix* mail server application to run on top of DOT with only 140 lines of code changes.

3. EARLY RESULTS

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 and name, file properties, etc. The bulk data transfer occurs via GTC-GTC communication.

We used this program to transfer files of sizes ranging from 4

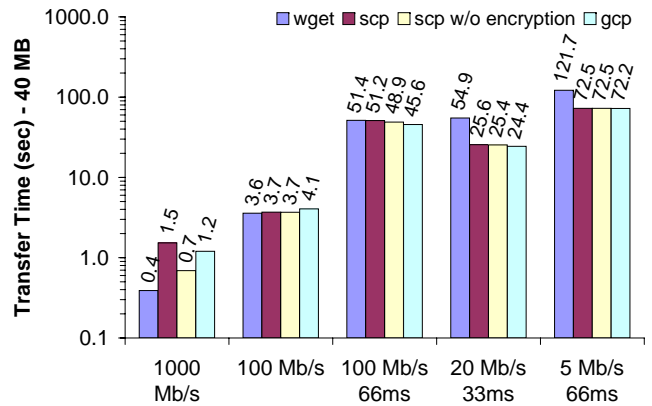


Figure 3: GCP Performance

KB to 40 MB under varying network conditions. In the interests of space, we present only the results from the 40 MB file transfer. These results, shown in Figure 3, 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 part. All displayed results are the mean of 10 trials. All experiments were performed on a Gigabit Ethernet network with a dummynet middlebox being used to control network bandwidth and latency.

gcp exhibits very little or no overhead when compared to the other file transfer tools in WAN conditions and performs significantly better than *wget* in the low bandwidth cases. Performance is equivalent to *scp* both with and without encryption. The only situation where *gcp* shows overhead is in the 1000 Mbit/s case. First, *wget* is the fastest as, unlike both *scp* and *gcp*, the Apache HTTP server is always running and *wget* does not pay the overhead of having to execute a remote process to accept the data transfer. Further, *gcp* is only slightly slower than *scp*. We examined the cause of this overhead and it is mainly due to fact that the GTCs on the sender side has to hash the data twice: once for the entire object and once for the descriptors that describe parts of the object. Most of this overhead can be eliminated by generating a cheaper version of the OID. For example, an equivalent OID could be generated by hashing the descriptors instead. However, this sacrifices an end-to-end consistency check as the OID is no longer a hash over the entire object. Given the low overhead of DOT in the WAN cases, we believe that retaining the end-to-end check is justified.