# An Improved Multiprotocol Application Data Transfer Service

Jude C. Nelson judecn@gmail.com

John H. Hartman jhh@cs.arizona.edu

April 14, 2012

Department of Computer Science

University of Arizona

Tucson, AZ 85721

**Abstract**

This paper presents the architecture, implementation and evaluation of IFTD, a multiprotocol data transfer service. The architecture allows a single implementation of a data transfer protocol to be used by any local application for its data transfer needs. This way, IFTD separates the development of application logic from data transfer logic, allowing application developers to avoid re-implementing data transfer protocols for each additional application. Also, applications that use IFTD will immediately benefit from additional data transfer protocols added to it.

This paper first provides an argument for the design considerations and requirements for creating IFTD based upon common features of real-world data transfer scenarios. It then gives a discription of how IFTD meets these requirements, and then assesses its effectiveness at performing data transfers in a series of four performance evaluations with respect to the performances of existing data transfer software. Finally, it discusses additional considerations that may be used to algorithmically determine the best data transfer protocol for given data and a given transfer history.

## 1 Introduction

IFTD is a data transfer service that provides a uniform interface for applications to transfer data via a variety of data transfer protocols [1]. An application that uses IFTD provides it with as much information as it has about the desired data (e.g. host, name, size, hash, etc.), as well as any additional information on how IFTD should use each available protocol (e.g. port numbers, login credentials, certificate authorities, etc.). If the

[1] In this paper, a *data transfer protocol* not only refers to any protocol conventionally used to transmit data between two or more hosts (e.g. HTTP, FTP, BitTorrent, etc.), but also any program or service IFTD can use to send and receive data (e.g. scp, CoDeeN, Gush, etc). Data transfer protocols exist within IFTD as pairs–a sender and a receiver.

application invokes IFTD to send data, it will additionally provide it with as much information as it has about the data itself.

If the local IFTD instance is able to connect to an IFTD instance on the remote host, both instances examine the data features, select a set of protocols available to both of them, and determine the best protocol for transferring the data. They then rely on the selected protocol to transfer the data from the source host (the host sending the data) to the destination host (the host receiving the data), but will select different protocols available to both in the event that the chosen protocol fails. If the local IFTD instance cannot contact a remote IFTD instance, it instead attempts to connect to the remote host with each protocol it supports until one is accepted, and then performs the data transfer via that protocol.

Currently, there are many data transfer protocols an application may use to transfer data, each with its own set of use cases [1]. Depending on the application's data transfer requirements, some protocols are preferable to others in certain cases, depending on the specific nature of the data. For example, downloading a web page to a single host is generally faster with HTTP than with BitTorrent, whereas downloading a software package to many PlanetLab hosts [14] from a single repository is generally faster with BitTorrent than HTTP [3], but neither HTTP nor BitTorrent are necessarily faster at sending short messages than IRC.

This variety poses a problem when designing new data transfer applications–the application developer must determine in advance which protocols to use for data transfer, and in which cases each protocol is applicable. It is intractable from an engineering standpoint to do this for every single data transfer application for several reasons. First, a change in the behavior of one protocol may require updating the protocol implementations in each application that uses it. Also, a change in the data requirements of an application could mean re-engineering part of or all of the application's protocol usages. Additionally, applications using the same protocols may inadvertently interfere with one another's ability to transfer data (e.g. one may require exclusive access to the same range of ports, blocking another's ability to use them). Finally, it can be difficult to design an application to tolerate protocol errors, depending on how error recovery would need to be implemented each time the protocol is invoked. These considerations only constrain an application's ability to use the best protocol for each data transfer.

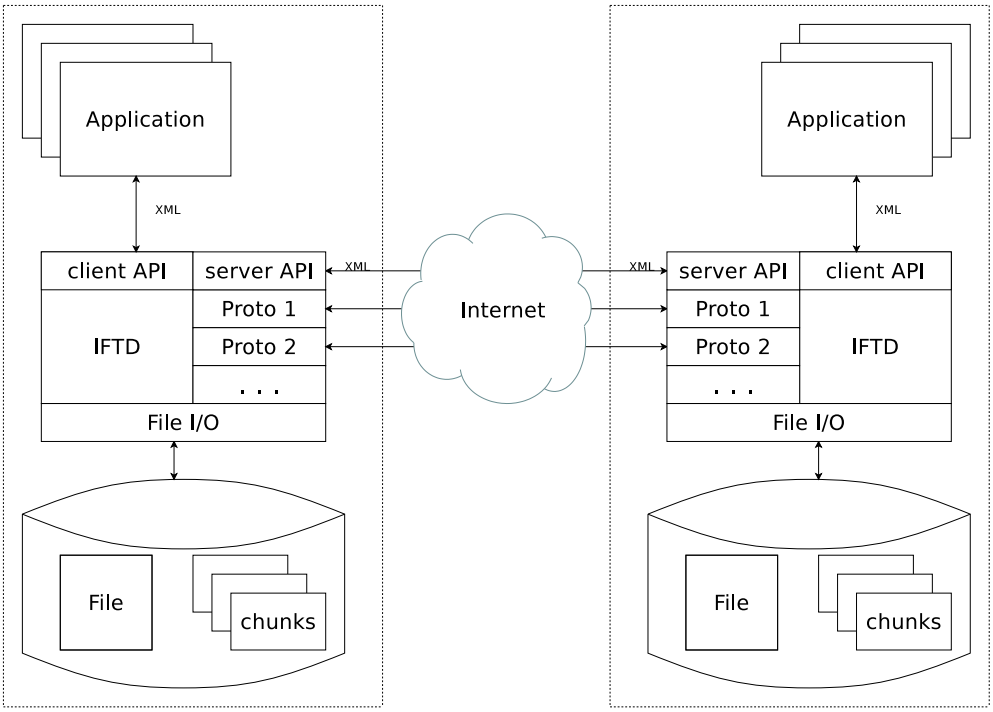


Figure 1: IFTD Overview

The solution to these problems is a data transfer service that intelligently selects the best protocol to transfer data, based on properties of the data itself and protocol usage information from the application. If sending data, it should make the data available to be pulled by a destination host via as many different protocols as possible along with the best protocol. If receiving data, it should also be ready to accept data pushed to it via any available protocol in addition to the best protocol. This removes the burden of designing applications to each use many different protocols; instead, the applications only need to use the data transfer service. Since the service identifies which protocols are best for transferring different classes of data, and prepares itself to communicate via as many different protocols as possible, the service reduces the application's burden of identifying which protocols should be used in which data transfer scenarios.

This paper presents a prototype implementation of such a data transfer service–the Intelligent File Transfer Daemon (IFTD). The use of IFTD for data transfers provides several opportunities to improve the quality of transmission that would otherwise be unavailable to individual applications. Since IFTD makes as many protocols as possible available for each data transfer, the transmission itself tolerates faults in individual protocols during the transfer. In the event that the transmission is interrupted or fails in one protocol, IFTD can select from its remaining protocols to resume transmission until either the transmission has completed or all protocols have been rendered unusable.

To implement automatic protocol fault tolerance, IFTD may break the data into fixed-length chunks

to be sent via its available protocols, making each protocol resumable regardless of whether or not the protocol originally supported it. This way, in the event that one or more protocols become unavailable during transmission, the destination host does not need to re-transfer data from the source host.

To increase bandwidth while receiving data, the receiving IFTD chooses the fastest protocol for transmission based on prior experience. It first acquires data features that influence transmission speed from the sending IFTD and optionally from the application. It then uses these features as input to a discriminatory classifier to identify the fastest protocol most frequently associated with data with similar features. It informs the sending IFTD of its decision, so the sending IFTD can subsequently use this protocol most frequently to send data. Once transmission is complete, the receiving IFTD identifies the fastest protocol used in this transfer and refines the classifier data with this protocol and the transfer data's features, this improving its classifier's ability to choose the fastest protocol in subsequent transfers.

The remainder of this paper is organized as follows. First, it presents a survey of previous work related to the ways in which IFTD solves the aforementioned problems. Next, it presents a top-down view of the architecture of IFTD as a multiprotocol data transfer service, given the context of the related work. It then presents an evaluation of IFTD's performance in comparison to two other multi-protocol data transfer frameworks: the file transfer framework arizonatransfer from Stork [3], and the `urllib2` module in Python 2.5. Finally, this paper discusses further work in refining the selection of "good" protocols for transferring data under a variety of circumstances.

## 2 Related Work

There are several bodies of work that explore improving data transfers between applications across a network. IFTD sets itself apart from these prior works by not only allowing an application to transfer arbitrary data, but also completely removing the need for an application to be aware of any data transfer implementations while simultaneously providing a data transfer architecture that supports the concurrent use of arbitrarily complex data transfer protocols.

The differentiation between content negotiation and data transfer within IFTD is borrowed from the Data-Oriented Transfer (DOT) architecture [20]. While DOT provides applications with the ability to transfer data without needing to implement their own data transfer protocols, it still requires that applications perform content negotiation directly. IFTD performs both tasks on behalf of its client applications.

IFTD's protocol architecture is inspired by three different existing architectures: the transfer plugin architecture in DOT, the protocol plugin architecture in the peer-to-peer file-sharing service giFT [9], and the package transport architecture arizonatransfer from the Stork package manager [3]. All three provide

data transfer protocol architectures that hide the details of the protocol implementation from the applications using them. Also, they both provide a degree of protocol fault-tolerance during transmission by transparently switching from a failed protocol to an operational protocol. Unlike giFT, however, the capabilities of IFTD's protocols are not constrained by a specific use-case, and unlike DOT, IFTD's protocol implementations may be internally non-resumable. Additionally, IFTD can receive data concurrently using different protocols, which is beyond the capabilities of arizonatransfer.

The way in which IFTD manages protocols to transfer data is inspired by the BASE methodology [17], is similar in design to the *mux conduits* and *protocol conduits* defined in the Conduits+ framework [10], and is similar in implementation to the way BitTorrent receives data [2]. Protocol behavior is dictated within IFTD by a high-level construct similar to a mux conduit called a *transfer processor*, which like BitTorrent implements a bitmap to represent which byte-ranges of the data have been transferred and which have not. Protocol implementations run concurrently and rejuvenate on every transfer to achieve a degree of transfer fault tolerance as recommended by the BASE methodology. Unlike Conduits+, however, IFTD's transfer processor is designed to manage application-layer protocols instead of transport-layer protocols. Unlike BitTorrent, IFTD handles scenarios where information that would otherwise be provided in a .torrent file, such as file size and chunk hashes, is not known in advance, and also implements a chunk-handling system that can optionally accommodate more data than expected and make guarantees about the chunk sizes and file reassembly that BitTorrent does not.

Finally, with regards to real-world applicability, IFTD instances are not limited to communicating with other IFTD instances. Like the abilities of the giFT and Slurpie [18] multiprotocol implementations, IFTD expects its protocol implementations to be able to communicate with alternative remote services well enough for the remote service to treat IFTD like a typical piece of client software.

# 3 Architecture

To understand IFTD's architecture, one must first consider what information is available to the sending and receiving hosts before they begin transmitting. IFTD is constructed around these five observations:

1. For any source/destination host pairing, neither host knows in advance which protocols the other will use to perform a transfer, but both know in advance which protocols are locally available.
2. A destination host may have performed data transfers similar to the pending one in the past.
3. A source host can perform measurements on the data in advance of sending it.
4. It is possible that no host knows everything about the data in advance.

5. One source host may carry out the data transfer with many destination hosts hosts. One destination host may carry out the data transfer with many source hosts.

These observations suggest that data transmission should occur in at least two stages, as with DOT: a content negotiation stage and a data transfer stage. A given destination host should not only acquire as much information as possible about the data to transfer, but also as much information as possible about the host(s) storing the data. This would include which protocols to use to receive the data, which source host(s) to contact for which pieces of the data, and how to know whether or not the data are being correctly transferred. Once a destination host knows these things, it will be better equipped to carry out the data transfer than if it had not made this attempt. A similar argument can be made about a given source host attempting to perform a data transfer to one or more destination hosts. Unlike DOT, however, an IFTD-aware application sends IFTD only the information it knows about the data transfer and about the hosts to engage so that IFTD can handle the content negotiation on its behalf.

The information IFTD requires from an application is packaged into an IFTD data structure called a *job*. A job represents all of the relevant information about a data transfer as a sequence of key/value pairs. The specific key/value pairs that the application presents are called *job attributes*, and are shared between IFTD instances to negotiate a data transfer. Key/value pairs that IFTD maintains internally to monitor the transfer are called *job statistics*, and are ultimately used by the classifier to associate classes of data with their best transfer protocols. The IFTD job attribute keys are summarized in Figure 3. When instantiated, a job will set default values for these keys.

An application may additionally provide parameters and hints to each protocol through *connection attributes*. The connection attributes are presented as key/value pairs that map protocol names to additional sets of key/value pairs, which in turn map protocol-specific connection attribute keys to appropriate values. Although the data are optional, some protocols require that certain connection attributes be specified in order for the protocol to be used.

### 3.1 IFTD Transfer Scenario

To help understand how applications use IFTD, consider this IFTD transfer scenario. Suppose Alice wishes to retrieve the latest version of the `nano` software package, and that Bob regularly makes this and other packages available under the `/tmp/pkgs` directory on his server. In order for Alice to retrieve the package, she can use one of several programs to communicate with Bob's server (e.g. `scp`, `wget`, `ftp`, etc.). Bob, however, regularly adds and removes services from his server, so at any given time some of these programs will not work. Since both Alice and Bob currently have an IFTD instance running on their machines, Alice decides

| IFTD Job Attributes | |
|---|---|
| JOB_ATTR_FILE_TYPE | MIME type of the file to transfer. |
| JOB_ATTR_FILE_SIZE | Size in bytes of the file to transfer. |
| JOB_ATTR_FILE_MIN_SIZE | Minimum allowable size for the file (if size is unknown). |
| JOB_ATTR_FILE_MAX_SIZE | Maximum allowable size for the file (if size is unknown). |
| JOB_ATTR_CHUNKSIZE | Fixed length of chunks during transfer. |
| JOB_ATTR_NUM_CHUNKS | Number of chunks in the file. This is used to cap the number of chunks to receive if the exact file size is not known. |
| JOB_ATTR_FILE_HASH | SHA-1 hash of the file. |
| JOB_ATTR_SRC_NAME | Path to the file on the source host. |
| JOB_ATTR_DEST_NAME | Path to the file on the destination host. |
| JOB_ATTR_SRC_HOST | Hostname or IP address of the source host. |
| JOB_ATTR_DEST_HOST | Hostname or IP address of the destination host. |
| JOB_ATTR_PROTOS | If given, this is a whitelist of protocols that IFTD may use. |
| JOB_ATTR_TRUNCATE | If given, IFTD will truncate chunks that are too long. |
| JOB_ATTR_STRICT_CHUNKSIZE | If set to True, any chunk with a length not equal to the value of JOB_ATTR_CHUNKSIZE will cause the transfer to fail. |
| JOB_ATTR_TRANSFER_TIMEOUT | Maximum amount of time the transfer may take. |
| JOB_ATTR_DO_CHUNKING | If set to False, IFTD will send the entire file as one chunk (this is True by default). |
| JOB_ATTR_MIN_BANDWIDTH | Minimum bandwidth any protocol must maintain while transferring (in Bps); otherwise the protocol is considered failed. |
| JOB_ATTR_MAX_ATTEMPTS | Maximum number of failures per protocol that IFTD will tolerate during chunk transfer. |
| JOB_ATTR_CHUNK_TIMEOUT | Maximum amount of time a receiving protocol may spend transferring a chunk. |

Table 1: Summary of built-in user-accessible IFTD job attributes, some of which are optional. They are determined by the application, and used by IFTD to carry out data transfers. Additional job attributes may be defined on a per-application and per-protocol basis.

to create and run a small application–a Python script–to retrieve the package using her IFTD instance. The application does not choose a specific protocol to use, but instead provides IFTD with some information about the data transfer so it can make the choice.

When Alice's local IFTD instance receives her application's transfer request via XMLRPC [19] for Bob's `nano` package, it examines each of its available protocols to determine which may be used, given Alice's information. It determines that she has given enough information to use its `http_receiver` or its `scp_receiver` protocols to download the package via HTTP or `scp` respectively. Alice, through her IFTD instance, sends these protocols, the application's job attributes (including the package's presumed location on Bob's server), and per-protocol connection attributes to inform Bob's IFTD instance that she wishes to receive the file.

When Bob's IFTD receives Alice's IFTD's request, it verifies that the package file exists and is accessable to it. It examines the protocols available to it and the protocols Alice's IFTD indicated that it can use, and initializes any sending protocols that wait for a receiving protocol to connect–in this scase, `http_sender`. It breaks the package file into a sequence of fixed-sized chunks, writes them to disk in a temporary directory, and informs the `http_sender` protocol of their location. Once `http_sender` is ready to receive connections,

Bob's IFTD replies to Alice's IFTD that it can use its `http_sender` and `scp_sender` protocols to send data, and provides the name of the directory in which the chunks reside. It also gives Alice's IFTD a set of file features to be fed into Alice's IFTD's protocol classifier, as well as the directory it will use to store chunks of the file and the hashes of each chunk.

When Alice's IFTD receives this reply from Bob's IFTD, it creates a feature vector representing the file features and uses it as input to its protocol classifier. Since other applications on Alice's workstation have invoked IFTD to perform data transfers in the past, the classifier calculates the probability of each available protocol having historically been the best protocol to transfer data with similar features. With this prior information, the protocol classifier finds that the `http_receiver` protocol has had the highest probability of being used to successfully transfer this type of data.

Now that Alice's IFTD knows which protocols Bob's IFTD supports, as well as the best protocol to use, it examines the capabilities of the `scp_receiver` and `http_receiver` protocols. Since `scp_receiver` can transfer data without needing `scp_sender` to be present, Alice's IFTD concludes that Bob's IFTD does not need to use its `scp_sender` protocol. Alice's IFTD informs Bob's IFTD of its intent to use `http_receiver` to receive data. It also gives Bob's IFTD the directory in which the chunks of the package will be received. Her IFTD does not inform Bob's IFTD that it will also be using `scp_receiver`, since it does not want his IFTD to use its corresponding `scp_sender`. Once Bob's IFTD receives this information, it waits until Alice's IFTD finishes receiving data via HTTP and scp, since it has been instructed to not use any of its active senders.

Eventually, Alice's IFTD finishes receiving the file, and sends acknowledgement to Bob's IFTD so it can shut down its sending protocols. Alice's IFTD finishes re-assembling the file from the chunks it received shortly afterward. It then records the feature vector it calculates from the file data and observes that the `http_receiver` protocol received data faster and more successfully than the `scp_receiver` protocol. Her IFTD refines its classifier's records with the feature vector along with the fact that `http_receiver` had the highest bandwidth, so that in the future her IFTD will be more likely to use the `http_receiver` protocol receive data matching the profile of the `nano` package.

It is not always the case, however, that both Alice's and Bob's IFTDs can always communicate with one another or have protocols in common. Suppose the next day that Alice tried to use her application to download `vim` from Bob's package directory, but Bob disabled his IFTD's `http_sender` and `scp_sender` protocols. Then, even though Alice's IFTD could contact Bob's IFTD, they could not agree on which protocols to use.

Fortunately for Alice, Bob's server also has a running Apache server that serves files from the package directory. Consequently, when the inter-IFTD content negotiation fails, Alice's IFTD falls back to simply using its `http_receiver` protocol to download the `vim` package from Apache on Bob's server.

```
# Alice's IFTD application

import iftapi
import sys

# package name is first arg
package = sys.argv[1]

job_attrs = {
  "JOB_ATTR_SRC_NAME"  :  "/tmp/pkgs/" + package,
  "JOB_ATTR_DEST_NAME" :  "/home/alice/" + package,
  "JOB_ATTR_SRC_HOST"  :  "cl31.cs.arizona.edu",   # Bob's server
  "JOB_ATTR_DEST_HOST" :  "localhost"

   # use large chunks, but not too large
  "JOB_ATTR_CHUNKSIZE" : 65536,
}

iftd_xmlrpc = iftapi.make_XMLRPC_client()
rc = iftd_xmlrpc.begin_ift(
   job_attrs, # job attribute dict
   None,      # connection hints
   False,     # not sending
   True,      # receiving
   4001,      # Bob's iftd xmlrpc api port
   "/RPC2",   # Bob's iftd xmlrpc directory
   60 )       # timeout after 60 seconds

sys.exit(rc)
```

Figure 2: Sample IFTD application that retrieves the given file from `cl31.cs.arizona.edu/tmp/pkgs` and stores it in `/home/alice`.

The following week, Alice attempts to download the `ex` package from Bob's server, but Bob had since uninstalled IFTD and Apache but is running an OpenSSH daemon on the server. Even then, Alice's IFTD still retrieves the `ex` package from Bob's package directory using her IFTD's `scp_receiver` protocol.

Even though all of these package transfer scenarios relied on different protocols and transfer services, Alice never needed to modify her application in these cases. Her application instead completely relies on IFTD to perform each protocol-specific content negotiation and data transfer, thus hiding from her all of the logic and implementation details for handling these different scenarios.

## 3.2 Chunk Handling

When using one or more resumable protocols or when communicating with a remote IFTD, IFTD sends and receives data in fixed-length chunks. When sending a file to a remote IFTD, it will read the file given to it by the application, break it up into chunks, and store the chunks within a temporary directory identified by

the file's base name and its SHA-1 hash. The receiving IFTD will create a corresponding temporary chunk directory on its host to store received chunks. The chunks are named in increasing numerical order based on which byte range of the file they contain, so that given the chunk size and file size, both an IFTD sender and receiver can identify which chunk names correspond to which pieces of the file. This number is called the *chunk identifier*. When IFTD finishes sending, it removes the chunks and the temporary directory. When IFTD finishes receiving, it reassembles the chunks into the original file and purges the temporary directory.

When receiving chunks, IFTD attempts to verify each chunk's integrity and writes each chunk to the appropriate offset within the file being received. Depending on the receiving application's job attributes, IFTD accepts, ignores, or truncates chunks that are not the correct size, and never re-receives the same chunk twice. If available, IFTD compares the SHA-1 hash of each chunk to the hash provided by the sender so it knows to re-download the chunk in case of data corruption. Once each chunk has been written and data transfer is finished, it will verify the hash of the re-constructed file against the file's known hash, if available.

In the event that a local IFTD attempts to receive data from a remote host and cannot communicate with a remote IFTD, it will instead use its protocols to attempt to get the file directly. If it uses resumable protocols like BitTorrent or HTTP, IFTD attempts to receive the file in fixed-length chunks to allow protocols to receive data concurrently and avoid receiving the same chunks twice. If no resumable protocols are available, IFTD treats the file as a single chunk and uses a non-resumable protocol to attempt to fetch it.

Breaking each file to transfer into chunks, which themselves are also files, offers IFTD-to-IFTD data transfers three advantages. Since IFTD protocols may not be resumable, this method allows IFTD to use non-resumable protocols to transfer part of the whole file. Also, combined with the fact that IFTD will allow each receiving protocol to transfer at most one unique chunk at a time, this method safely allows any protocol to transfer chunks of the original file concurrently. Additionally, creating chunks allows IFTD to present both the chunk data and the path to the chunk on disk to its sending protocols. This last advantage is useful in implementing IFTD protocols which depend on 3rd party transmission software such as `scp` or `netcat` that may require that the data to send be presented to it exclusively as a byte array or as a file path.

## 3.3 IFTD Content Negotiation

Given observations (4) and (5), the receiving IFTD acquires as much useful information as possible about the file to transfer in order to optimize the transfer process. For a given IFTD instance, there are three possible ways for this to occur: the receiving IFTD requests one or more remote IFTD instances to begin transferring data to it, the receiving IFTD instance is contacted by another IFTD instance wishing to send data to it, or the receiving IFTD requests a file from one or more remote hosts on which there are no IFTD instances.

These are the *Receiver Startup*, *Sender Startup*, and *Lone Receiver* scenarios, respectively.

### 3.3.1 Receiver Startup

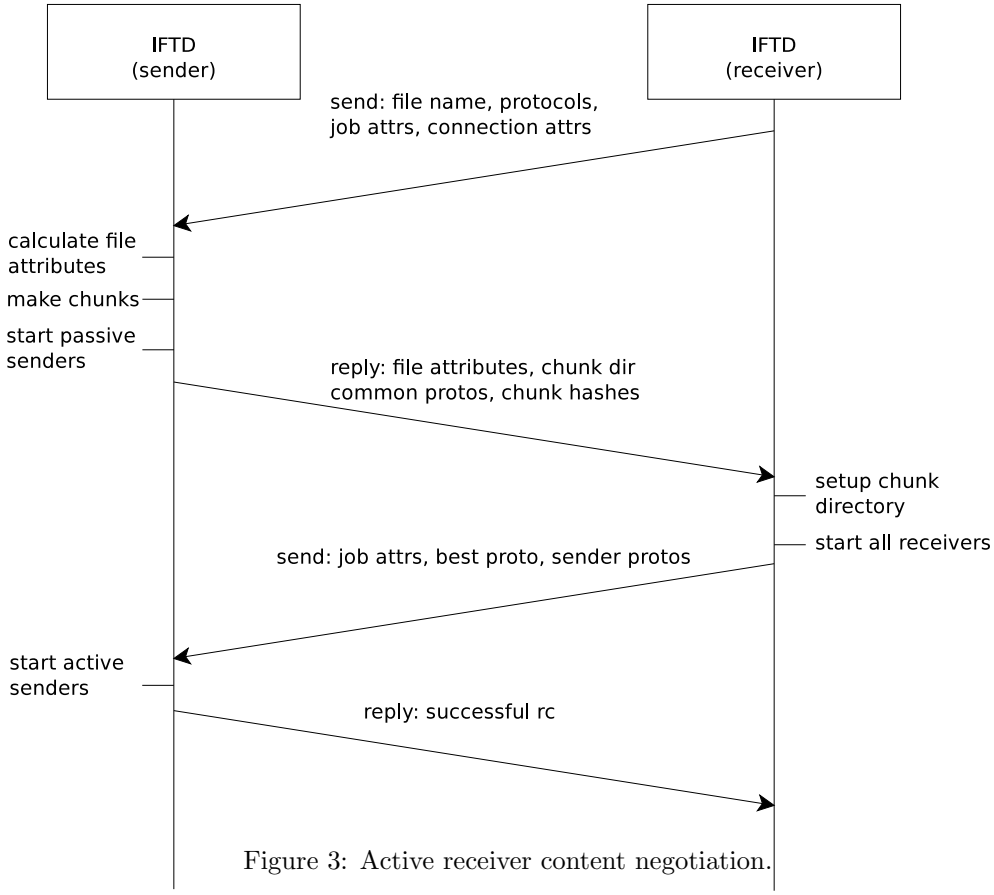


Figure 3: Active receiver content negotiation.

In this scenario, an application makes a request to the local IFTD instance to receive data from a remote IFTD instance on its behalf, similar to Alice's application in the example. IFTD begins negotiations by verifying that the path keyed by `JOB_ATTR_DEST_NAME` is writable. If so, it sends this path, a list of all of its supported protocols, the application's job attributes, and the application-given protocol connection attributes to the remote IFTD instance. If the remote IFTD can read the path keyed by `JOB_ATTR_SRC_NAME`, it first attempts to initialize any protocols that need to wait until a receiver connects, if any are deemed usable. It creates a record of the pending transfer, and assign it a timeout. It then breaks the file into chunks and puts them in a temporary directory. Finally, it sends back a list of protocols supported by both IFTDs, as well as the file size, MIME type, SHA-1 hash of the entire file, SHA-1 hash of each chunk, and the path to the temporary chunk directory it will create. If the application provided any information about the file that contradicts the remote IFTD's information, the transfer is aborted. If the timeout passes before content

negotiation completes, the transfer is aborted as well.

If the local IFTD instance accepts the information, it will initiate the transfer. Using the information given to it, it checks that the path keyed by `JOB_ATTR_DEST_NAME` refers to a writable location. If so, it makes its temporary chunk directory, initializes a receiver protocol for each protocol both IFTDs have in common (and are applicable), and identifies the best protocol with which to transfer the data. If not, the transfer is aborted.

The best protocol for the transfer may not be common to both IFTD instances since the classifier considers all previous transfers in making its decision. Regardless of whether or not the best protocol can be used, and as long as there is at least one usable protocol and the destination path is acceptable, the local IFTD requests that the remote IFTD start sending data to it since it now knows what to expect and how to receive it. The request includes the local chunk path, the choice for the best protocol (or a sentinel value indicating that the best protocol is unavailable), and a list of protocols that were successfully initialized and the sender should attempt to use. As shown in the example of Alice and Bob, the local IFTD does not tell the remote IFTD to start any protocol that will send data without the local IFTD's oversight.

Once the remote IFTD receives this information, it initializes all protocols that do not require a receiver to connect and prepares to send data. It will create its temporary chunk directory, split the file into chunks and populate its temporary chunk directory with them, initialize the protocols the receiver indicated it should use, and finally acknowledge the local IFTD. This completes the content negotiation in this scenario.

### 3.3.2 Sender Startup

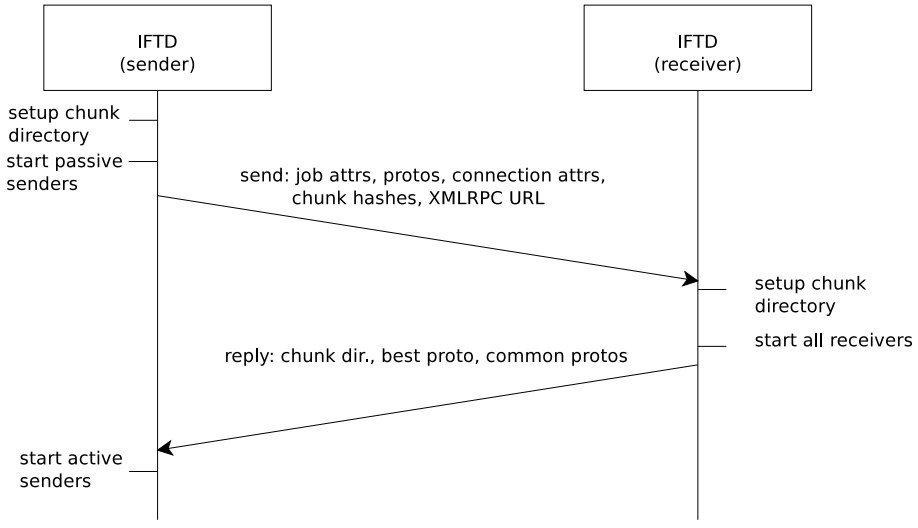


Figure 4: Active sender content negotiation.

In this scenario, an application makes a request to the local IFTD instance to send data to the host

indicated by the JOB_ATTR_DEST_HOST job attribute. Using the Alice and Bob example, in this scenario Bob pushes a package to Alice. To begin the content negotiation, the local IFTD first ensures that the path keyed by JOB_ATTR_SRC_NAME is readable. If so, the local IFTD determines the prerequisite file metadata the remote receiving IFTD needs in order to begin transferring data–the file's MIME type, SHA-1 hash, size, chunk hashes, and temporary chunk directory, as well as the job attributes from the application and any connection attributes the application provided. It breaks the file into chunks, and sets up all of the sender protocols that need to wait for a corresponding receiver to connect. It makes the chunk data available to each of these. Then, it attempts to send the file metadata to the remote IFTD receiver. If the remote receiver does not respond, the transfer is aborted, the protocols shut down, and the chunks removed.

When the receiver gets this information, it checks to see that the path keyed by JOB_ATTR_DEST_NAME is writable. If so, it creates its temporary chunk directory and starts receiving with all protocols it has in common with the sender; if not, the transfer is aborted. If at least one protocol successfully initializes, the receiver uses the information about the file as input to its protocol classifier. It will respond to the sending IFTD with the best protocol choice (or a sentinel indicating that the classifier's choice was not a common protocol), the protocols that it successfully initialized, and its chunk directory.

Upon receiving this acknowledgment, the sender initializes its active sending protocols. This completes the content negotiation in this scenario.

### 3.3.3 Lone Receiver

If the receiving IFTD cannot contact a sending IFTD, it assumes that there is no remote IFTD instance. Consequently, during data transmission it will only know any information about the data that the application gave it initially, and it may gather information from remote hosts based on the capabilities of the available protocols. The directory on the remote host containing the file is treated like the sender's temporary chunk directory, and depending on the capabilities of the available protocols, the file itself may be treated as a single chunk if none of the protocols are resumable. As with the receiver startup scenario, IFTD will only receive the file to a path under the directory it is configured to use to store files it receives.

To perform the transfer, IFTD attempts to use each of its available receiver protocols until one successfully downloads the file. If available, IFTD intelligently uses protocols that can perform partial data requests, such as HTTP and BitTorrent, to request different parts of the file to maximize bandwidth. Once the receiver completes the transfer, it uses any of the (otherwise sender-provided) information it was given by the receiving application to attempt to verify the integrity of the file.

### 3.4 Data Transfer Protocol Architecture

IFTD protocols are implemented as Python objects which inherit functionality from abstract IFTD-provided Python classes that drive the transmission process. IFTD defines abstract classes for a sender and a receiver, and both are considered to be distinct protocols. This means that `http_sender` is a different protocol from `http_receiver`.

IFTD protocols may use any means necessary to exchange data. For example, the default IFTD source package includes protocols that communicate directly through a TCP socket, use an HTTP server and client to download chunks, run the `scp` program as a shell subprocess to transfer files, get data from a local `squid` cache, and join a BitTorrent swarm get chunks using the Rasterbar libtorrent API [13]. IFTD protocols may even invoke IFTD recursively to perform a transfer, or invoke additional multiprotocol transfer frameworks on IFTD's behalf.

IFTD makes a distinction between active and passive protocols as well as sending and receiving protocols. An *active protocol* is a protocol that drives the data transfer and is responsible for triggering data movement. For example, IFTD's `http_receiver` protocol is an active receiver, since it triggers data transmission from an HTTP server by sending it a GET message. Conversely, a *passive protocol* is a protocol that waits for its counterpart to initiate the data transfer. In this example, the HTTP server is a passive sender, since it makes data available to be served but does not actually move the data without receiver intervention. Both sender and receiver can be active, such as the scp-driven `scp_sender` and `scp_receiver` protocols, and both sender and receiver can be passive, as with BitTorrent-driven protocols `bittorrent_sender` and `bittorrent_receiver`.

IFTD additionally identifies protocols based on how well it can use any inherent resumability in the protocol (it's `chunking capability`). Protocols like HTTP, which can send and receive given byte ranges of a file, are known to IFTD as *deterministic chunking protocols* (DCPs). That is, IFTD identifies one or more one or more chunks in the file to transfer, and the protocol transfers only those specific chunks without having to rely on one of the IFTDs to break the file into chunks for it. Protocols that can still send and receive chunks of a file but do not allow IFTD to choose which chunks are known as *nondeterministic chunking protocols* (NCPs). This includes protocols such as IFTD's default BitTorrent sender and receiver and raw TCP socket receiver, since the protocol, and by extension IFTD cannot know in advance which chunks will be transferred. This distinction is important for managing transmission in that consideration of a protocol's chunking capability can be used to avoid duplicate chunk transfers.

Each IFTD protocol has a lifespan of four stages: its one-time setup stage, its per-transfer setup stage, its data transfer stage, and its clean-up stage. The last three stages are invoked for each data transfer, and

for receivers last two stages occur within a separate thread from the main IFTD thread to help protect the software from a fatal protocol error or timeout. The flow control of each stage is summarized in Figure 5.

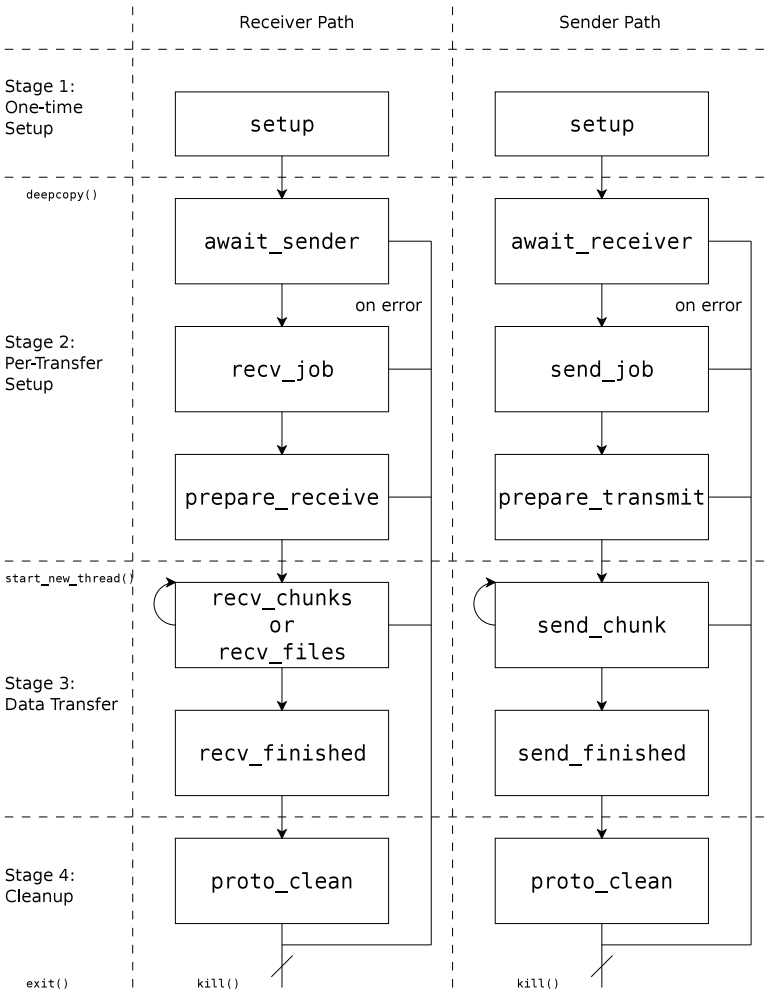


Figure 5: Lifecycle of a sender and receiver protocol within IFTD.

### 3.4.1 Protocol One-Time Setup

Before a protocol can be used, it must be instantiated for the first time when IFTD starts up and then given a chance to perform any protocol-specific one-time configuration. The `setup(setup_attrs)` method is responsible for this and is called by IFTD once the first protocol instance is loaded into memory. For example, the `http_sender` protocol that comes with IFTD will start an HTTP server when its `setup` method is invoked. The `setup_attrs` argument is a Python dictionary that maps protocol-specific connection attributes to meaningful values, and is constructed from IFTD's configuration file.

Once a protocol object has completed its one-time setup, IFTD will keep a single instance of it in memory

and, through virtue of Python's `copy` module, create a copy of it to use for each subsequent transfer. This way, protocol instances do not need to be designed to be reusable–once a protocol's copy completes one transfer, the copy is destroyed.

### 3.4.2 Protocol Per-Transfer Setup

The per-transfer setup stage in each protocol is invoked on a copy of a protocol. This stage is responsible for carrying out that protocol's content negotiation. The process is divided into three steps: acting on the application's job, setting up a connection, and preparing to transfer chunks. For senders, the three methods that carry out these steps are:

```
send_job(job)
await_receiver(connection_attrs, timeout)
prepare_transmit(job)
```

For receivers, these methods are:

```
recv_job(job)
await_sender(connection_attrs, timeout)
prepare_receive(job)
```

The purpose of the first step is two-fold. First, it gives the protocol a chance to extract any useful information that the application provided in its job attributes so it can later negotiate with the remote host. This way, the protocols can raise errors if any of the given data are malformed or missing before further initialization occurs. Second, this step allows the protocol to add any additional information to the job for other protocols to use, since the job is shared between each protocol and the IFTD transmission logic. For example, if the job does not provide a file size, the `http_receiver` protocol will attempt to contact the remote HTTP server to query the size and store it in the job for other protocols to use.

Once it has its job information, a protocol may need to set up a connection to a remote host before it can prepare to transfer chunks. This step may be a blocking operation. For example, the TCP socket receiver protocol uses its `await_sender` method to start a server socket and wait for the sender to connect to it before it can begin receiving data. Both senders and receivers are passed a timeout value that they are expected to honor. Protocols are not required to implement these methods, however, if this step is not needed.

The final step in protocol-specific content negotiation finishes any remaining per-transfer setup tasks, such as checking the job for any additional information supplied by other protocols. For example, the `bittorrent_sender` protocol uses its `prepare_transmit` method to create a torrent file for the data it will begin to share. Protocols are not required to implement these methods if there are no remaining setup tasks to perform.

### 3.4.3 Sending Data

Once each protocol finishes its negotiation, IFTD begins to use them to transfer data. In sending protocols, sending data is accomplished by two methods:

```
send_chunk(chunk, chunk_id, chunk_path, remote_chunk_path)
send_finished(status)
```

The first method is called repeatedly to send individual chunks, and the second method is called within the protocol to indicate that the protocol can no longer send chunks and should be given a chance to shut down. The `chunk` and `chunk_id` arguments are a byte array containing the chunk data and the numerical chunk identifier, respectively. The `chunk_path` and `remote_chunk_path` arguments are the path to the chunk on disk to send and the path on the remote host's disk where the chunk is to be sent. The first method returns the number of bytes sent. Returning 0 is interpreted to mean that the protocol can no longer send data, but its transfer status should not be marked as a failure.

Depending on whether or not the protocol is a passive sender, `send_chunk` may not actually write data over the network. For example, the `http_sender` protocol, since it is passive, accumulates the arguments given to this method over time to determine which chunks on disk it is allowed to serve to the active `http_receiver` protocol. In the case of the `bittorrent_sender`, this method does absolutely nothing since the BitTorrent library it uses performs the actual transmission without IFTD's interference.

### 3.4.4 Receiving Data

When a receiving protocol is transferring the application's data, it uses thread given to it by IFTD to continuously attempt to receive chunks. The methods that control this stage of life are analogous to those of a sending protocol:

```
recv_chunks(remote_chunk_dir, desired_chunks)
recv_files(remote_file_paths, local_file_dir)
add_chunk( chunk_id, chunk_bytes )
add_file( chunk_id, chunk_path )
whole_file( file_path )
recv_finished(status)
```

IFTD receiver protocols may receive data either as strings of bytes or as files, depending on whether or not the protocol is inherently resumable. Resumable protocols are expected to implement the `recv_chunks` method, while non-resumable protocols are expected to implement the `recv_files` method. The method IFTD uses to drive transmission is selected and used exclusively for the duration of the transfer once this stage in its lifecycle is entered, based on the protocol's chunking capability.

In `recv_chunks`, `remote_chunk_dir` refers to the path to the temporary chunk directory on the sender's disk, and `desired_chunks` is a list of chunk names IFTD needs to receive. If the protocol is resumable, this method will be called repeatedly until the protocol implementation invokes the `recv_finished(status)` method to signal to IFTD that it can no longer receive. It passes the chunks it receives to IFTD via the `add_chunk` method.

Analogously, non-resumable protocols use the `recv_files` method to receive data as files instead of as byte streams. The `remote_file_paths` argument is an array that identifies the remote path to each file on the remote sender, as well as which chunk it represents. `local_file_dir` represents the location on disk to which to write any received files.

Since non-resumable protocols may be used with or without the presence of a remote sending IFTD, there are two methods a non-resumable protocol uses to identify the data it receives to IFTD. The former–`add_file`–is used to identify the location on disk of a chunk of the larger file being received. These chunks are ultimately read by IFTD when it reassembles the chunks into the whole file. The latter method–`whole_file`–is used to inform IFTD in the event that the protocol receives the file in its entirety. This method may be used when there is no remote IFTD sender and a non-resumable protocol succeeds in downloading the entire file all at once.

To ensure that multiple receiving protocols do not download the same chunks, IFTD implements a global chunk reservation system as part of its file I/O subsystem. This allows a protocol to temporarily gain exclusive access to a given chunk. While a chunk is reserved, no other protocol will intentionally receive it.

If there is a remote IFTD sender available, or the protocol is a DCP, the protocol first queries the reservation system for an unreserved chunk. The protocol continuously delays and tries again if no chunks are available and the file has not yet been reassembled. When it succeeds in reserving a chunk, it records how long the reservation lasts (given in `JOB_ATTR_CHUNK_TIMEOUT`) until another protocol can reserve it. This way, if the protocol hangs during transmission, another protocol can re-reserve and download the chunk instead. Once a chunk has been reserved, the protocol proceeds to transfer the chunk to the chunk directory.

If the protocol is an NCP, it does reserve a chunk in advance since it does not know which chunk will be transferred. Any attempt to do so would otherwise carry the risk of inadvertently reserving chunks that the NCP will not write, blocking other protocols from reserving them and thus decreasing the effective bandwidth. Instead, an NCP receives its chunk first, and then attempts to store it in the file if there is not yet any data for the given chunk.

When the transfer completes, the protocol atomically checks to see that there are no data written in the file for this chunk and if not, it writes the data to the file. Once the chunk has been written, the protocol atomically releases the chunk and marks it as downloaded. Then, subsequent writes or reservation requests

for that chunk will be rejected by the reservation system. Transmission continues in this manner until all chunks are received, or until one protocol manages to download the entire file all at once.

The chunk reservation system is unused in the event that there is no remote IFTD and that the only protocols available are all non-resumable. In such a case, using the reservation system adds needless overhead to data transfers since the only way to transfer the file in this scenario is to try each protocol until one performs the whole transfer or they all fail.

In the event that the receiving IFTD does not know the file size, the chunk reservation system will dynamically add more chunks to the file as they are received. That is, if a protocol receives a chunk that logically occurs $k$ chunks after the last chunk of the file, the chunk reservation system will add mud site:7chan.org$k$ additional chunks to the file that other protocols may reserve. It will not allow protocols to reserve chunks that are beyond the maximum allowed size of the file.

### 3.4.5 Protocol Shutdown

Because protocols define both a one-time setup and per-transfer setup procedure, they must additionally define a per-transfer shutdown and one-time shutdown procedure. To shut down a single transfer, IFTD invokes a protocol's `proto_clean()` method in order give it a chance to release any global resources held during transmission before the protocol data are freed. This includes temporary files, pipes, sockets, etc. To perform a one-time shutdown, IFTD invokes a protocol's `kill()` method to give it a chance to permanently stop transmission. This method is called when IFTD itself shuts down. For example, the default `bittorrent_receiver` protocol will leave the BitTorrent swarm when its `proto_clean()` method is called, but will shut down the Rasterbar `libtorrent` library when its `kill()` method is called.

## 3.5 IFTD Data Transfer

Data transmission in IFTD is driven by singleton entity called the transfer core. Its purpose is two-fold– it manages and maintains data on all active transmissions, and it separates the content negotiation code (including the XMLRPC API) from the data transmission processing. The XMLRPC server in IFTD passes the data it acquires during negotiation to the transfer core, so the transfer core can drive the transmission process in a separate thread while the XMLRPC server handles additional application requests and content negotiations.

As a security measure, IFTD maintains two thread pools from which the transfer core acquires worker threads to drive transmissions. One pool is for sending threads, and the other is for receiving threads. If a thread pool is full, subsequent requests for threads result in causing the pending transmission to be aborted.

This limits the ability of a malicious user to deny the services of IFTD and its host server from legitimate users, and limits the damange a buggy application can do by requesting too many transfers at once.

### 3.5.1 Sending Data

Regardless of which protocol the receiver deemed the best protocol to use, the transfer core in the sending IFTD makes all chunks available to passive senders. This is because passive senders do not actually send data across a network until instructed to do so, but instead make it available to their receiver counterparts. Consequently, there is little if any bandwidth cost associated with sending data with passive sender protocols[2], and sending all chunks through all passive senders allows any active receiver protocol to perform data transfers for the receiving IFTD.

Since active sender protocols attempt to write data across a network, the transfer core selectively uses active sender protocols to send chunks. In this case, the transfer core sends chunks with an active sender's `send_chunk` method until the entire file is sent or the protocol encounters an error. If there was an error, the transfer core falls back to using a different active sender protocol it has in common with the receiver, and continues to fall back on the remaining active sender protocols until every protocol has failed more times than the application's tolerance for protocol error. The best protocol, if it is available and is an active sender, is the first protocol attempted. If the receiver does not give a preferred protocol, the receiver cycles through the active protocols, given each of them a chunk in turn, until the file is sent or each of them has failed more times than the application's tolerance for protocol errors.

### 3.5.2 Receiving Data

When IFTD begins to receive data, it creates a separate thread for each receiving protocol to use. This is done for two reasons. First, this way every passive receiver protocol can wait for an active sender concurrently, minimizing the odds that an active sender will send data in vain. Second, with or without a remote IFTD, letting every active receiver request and save different chunks from one or more source hosts concurrently results faster data transfer rates than receiving from only one protocol at a time in the event that certain protocols maintain higher bandwidth than others. The global chunk reservation system facilitates this independently of the transfer methods within the protocols' implementations.

In order to determine whether or not a given transfer succeeded, the transfer core monitors the transmission states of all running protocols for that transmission. If no protocols are running and the file is incomplete, or all protocols have failed, then the transfer itself fails. If one or more protocols indicate that

[2]The exception to this are NCP protocols such as BitTorrent, but since in implementation IFTD does not know in advance which, if any, chunks it will send, IFTD naively assumes that it will not send data until a peer requests data from it.

there are no more chunks to reserve in the event that the file size is known, or all protocols cease transferring without any indication of error in the event that the file size is unknown, then the transfer succeeds since this means that all chunks have arrived.

While each protocol receives data, the transfer core periodically calculates each protocol's bandwidth. If the bandwidth drops beneath an application-specified minimum bandwidth, the transfer core will stop the protocol. This is advantageous since the Python global interpreter lock severely limits the ability of multithreaded Python software to achieve true concurrency without resorting to running multiple Python virtual machines or invoking 3rd party libraries and/or blocking operations [15]. Additionally, killing slow protocols allows a transfer to fail in a reasonable amount of time in the event that the network between the receiving IFTD and some or all of the source hosts is inadvertently or intentionally congested.

The receiving transfer core additionally monitors the total effective data transferred between all protocols. If the total amount of data received exceeds the application-specified maximum allowable file size, the transfer fails and all protocols are stopped. This prevents the scenario where the destination host does not have enough space for a file with an underestimated size, and the scenario where a malicious source host continuously sends data with the intent of filling the destination host's storage to capacity [4].

## 3.6 Protocol Classification

The features of the data that are measured to help IFTD choose the best protocol are the success/failure of the data transfer, the MIME type of the data, the approximate size of the data, and the approximate time of day at which the transfer completed. These features are chosen based on prior real-world events, and are intuitively likely to affect the best protocol choice. For example, some of the more popular Linux distributions offer several different ways of downloading their ISO images, such as HTTP, FTP, BitTorrent, and Jigsaw Download [11], and cite protocol performance for large files as a consideration in making the choice [7, 8, 21]. Consequently, IFTD considers approximate file size when choosing the best protocol to transfer data. The intuition that the MIME type possibly affects the best protocol choice is not without precedent either, since some file servers have the ability to throttle data based on MIME type [12]. The time of day is considered relevant since the number of users are actively transferring data at a given time changes based on when people are likely to use their computers throughout the day [6].

When a protocol transfers a chunk, IFTD will record the protocol's name, the status (success or failure) of the transfer, the start and end times of the transfer, and the number of bytes transferred. IFTD accumulates this data within the job statistics, so when the transfer completes IFTD will have a record of every chunk transferred. When a transfer finishes, IFTD ranks the protocols in order from highest bandwidth to lowest

bandwidth, as calculated from the per-chunk transfer records created by the protocols. Then, IFTD calculates a feature vector from the re-assembled file that contains the file's MIME type, the file's approximate size, the approximate time of day the transfer occurred, and whether or not the transfer was successful. It combines the feature vector with the protocol with the highest bandwidth to produce a labeled feature vector, which it logs into a temporary buffer that, when full, will be used to refine the protocol classifier before being emptied.

IFTD uses a *naive Bayes classifier* to determine the best protocol to transfer data when given its feature vector. A naive Bayes classifier is a probabilistic classifier that uses Bayes' Theorem to select protocol $b$ that maximizes $p(b|X)$, where $X = (x_0, x_1, ..., x_n)$ is the data feature vector [16]. In other words, the classifier calculates $p(b|X) = p(X|b)p(b)/p(X)$ for all protocols $b$ and selects $b$ with the maximum $p(b|X)$. It is considered to be naive because it assumes that $x_i$ is independent of $x_j$ whenever $i \neq j$. Consequently, it calculates $p(X|b)$ as the joint probability $\prod p(x_i|b)$. It calculates values for $p(x_i|b)$ using the labeled feature vectors IFTD accumulates with each successive transfer.

There is a trade-off between how detailed the feature vector can be and how useful the values of $p(x_i|b)$ are. As the granularity of the data increases, the number of unique feature vectors increases, and the values of $p(x_i|b)$ cluster near 0 consequently since there are less cases where $x_i$ more than once given $b$. Conversely, as the granularity decreases, the number of unique feature vectors decreases and the values of $p(x_i|b)$ increase and cluster near 1 as a result since there are more cases where $x_i$ occurs multiple times given $b$. This is why IFTD records an approximate file size and an approximate time of day instead of their more precise measurements [3].

# 4 Evaluation

This section presents results and interpretations of four experiments designed to evaluate the performance and behavior of IFTD. The questions explored by these experiments include:

1. How well can IFTD tolerate protocol errors in the active receiver scenario when the best protocol is unknown?

2. How much overhead does a lone receiver IFTD introduce into data transfers relative to other similar multiprotocol data transfer software?

3. How well can a lone receiver IFTD, using DCPs, maximize effective bandwidth relative to other, similar multiprotocol data transfer software?

[3] In practice, IFTD arbitrarily records the file size as the multiple of $2^{16}$ bytes closest to the actual size, and the time of day as the multiple of 3 hours closest to the actual time. These values were chosen to keep the number of different $x_i$ values for size and time on the order of 10

4. If given time to learn, does the naive Bayes classifier in a lone receiver IFTD correctly identify the best data transfer protocol for prior labeled data feature vectors?

These experiments were run on two single-processor machines on a gigabit copper switched LAN with no other network traffic present. Each machine had an Intel® Pentium® 4 CPU clocked at 2.4 GHz with 2 GB of RAM, of which 1.25 GB was allocated to a RAM disk. Each machine had an Intel(R) 82540EM gigabit Ethernet controller. On each machine the files to transfer and the IFTD chunk directory were stored on the RAM disk so that the time-dependent experiments would not be affected by disk I/O operations.

## 4.1 Protocol Fault Tolerance

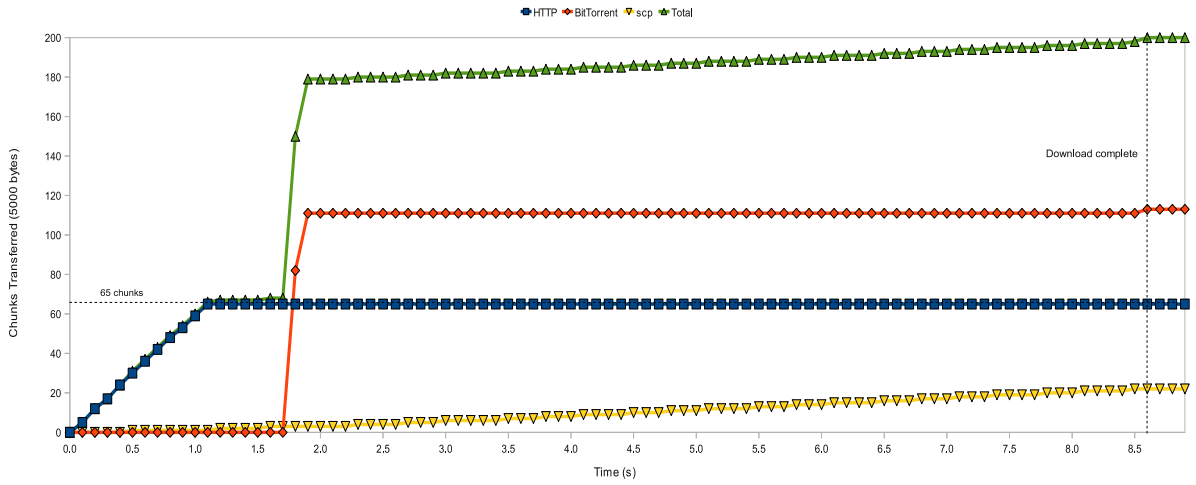


Figure 6: Number of chunks transferred by each protocol as a function of time. The HTTP protocol is programmed to transfer 65 chunks and fail, while the BitTorrent and scp protocols continue to receive the remaining 135 chunks.

An important aspect of a multiprotocol data transfer service is how well it can tolerate errors in single protocols. The purpose of this experiment is to demonstrate how well IFTD can tolerate protocol failures by measuring how many chunks each available protocol transfers at different times during a transfer. For this experiment the local IFTD is configured to use three protocols to receive a million-byte file that has been broken into 200 evenly-sized chunks by a remote IFTD. The first protocol uses the `scp` binary to retrieve chunks from the remote host. The second protocol uses the Rasterbar libtorrent library [13] to retrieve pieces of the file from a BitTorrent swarm, where the remote IFTD is a seed for the file. The third protocol uses `http` to retrieve chunks from the remote host, but is programmed to arbitrarily fail after receiving 65 chunks.

The protocol usages are summarized in Figure 6. Upon execution, the HTTP protocol will reserve a chunk, retrieve it, write it, and repeat the process 65 times before failing. The `scp` protocol runs concurrently

with the HTTP and BitTorrent protocols, but since it must perform public/private key encryption to begin receiving each chunk, it receives data at a much slower rate than the other two. The BitTorrent protocol takes longer to start receiving–longer than it takes the HTTP protocol to receive all of its pre-programmed chunks–but once it learns the IP address of the remote host from the BitTorrent tracker, it quickly receives most of the data and concurrently stores them alongside the `scp` protocol. It contacts the remote IFTD periodically for more chunks and receives a few more at nearly 8.4 seconds into the transmission.

The local IFTD does not have any prior information with which to train its classifier, so it receives data using all three protocols concurrently. Consequently, the behavior demonstrated here is typical only when IFTD has not performed enough transfers to initialize its classifier.

Since the BitTorrent receiver implementation is an NCP and does not know in advance which chunks will be sent to it, it receives many duplicate chunks relative to the HTTP and `scp` receivers. In fact, it received two chunks that the `scp` protocol, a DCP, had reserved and was in the process of downloading. The duplicated effort resulting from using BitTorrent alongside HTTP and `scp` is summarized in the following table.

| | HTTP | BitTorrent | `scp` |
|---|---|---|---|
| Chunks Saved | 65 | 113 | 22 |
| Chunks Re-downloaded | 0 | 86 | 2 |

Table 2: How many chunks were written versus how many chunks were rejected by the chunk reservation system using the protocols in Figure 6.

Duplication notwithstanding, this experiment demonstrates that IFTD can tolerate a complete protocol failure and continue to receive data. The application for which IFTD performs a transfer does not need to implement any logic for handling such errors.

## 4.2 Transfer Overhead

One consideration in choosing a multiprotocol data transfer service is the measure of how much overhead each service adds to an application's transfer processing. This experiment demonstrates how fast IFTD retrieves a file from a remote host relative to arizonatransfer and `urllib2`. All three pieces of transfer software retrieved files from a CherryPy 3.1.2 HTTP server [5], and were given the file's size in advance. IFTD used only its HTTP protocol, received only from CherryPy, and retrieved each file as a single chunk[4].

The data in Figure 7 represent the average of 10 consecutive transfers for each piece of transfer software for a range of file sizes. The bandwidth was calculated as the file size divided by the amount of time spent setting up, receiving data from, and shutting down a connection to the remote host. In IFTD, the

[4]In implementation, this allowed IFTD's HTTP protocol to avoid performing byte-range GETs, making it comparable to arizonatransfer and `urllib2`.

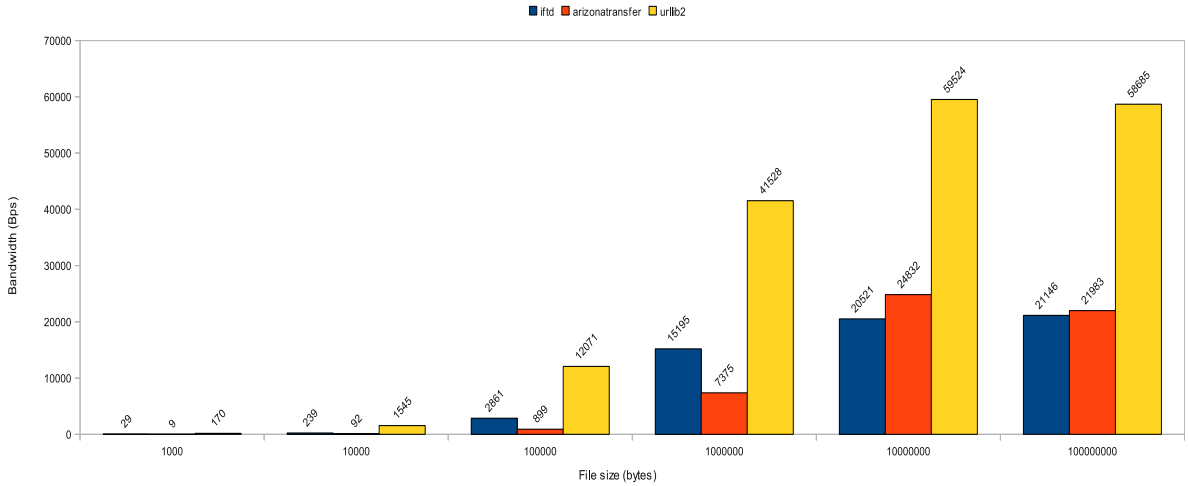


Figure 7: Bandwidth as a function of file size. Because all three pieces of software transfer data across a gigabit LAN, the affect of transfer overhead on each piece of software's bandwidth is noticeable for files less than 10MB in size.

amount of time measured is the round-trip time for an XMLRPC call to IFTD's `begin_ift()` method. In arizonatransfer, the amount of time measured is the time taken for the call to the `getfiles1()` method to complete. In `urllib2`, the amount of time measured is the time taken for opening a new file, calling the `urlopen()` method to contact the remote host, calling the `read()` method on the `Response` object returned by `urlopen()` to receive all the data, writing the data to the file, and closing the file. In the first two cases, file integrity checking is disabled.

Predictably, `urllib2` retrieved all files faster than arizonatransfer and IFTD. This is because the method used to receive data in a `urllib2` `Response` object is mapped directly to the Python socket package's `recv()` method [22], which in turn calls the POSIX `recv()` function with the actual socket descriptor in Linux to receive data from the remote host. Also, unlike arizonatransfer and IFTD, an application that uses `urllib2` supplies the protocol and URL that identify the way in which to receive data, thus freeing `urllib2` from the responsibility of determining which protocol to use. Consequentially, there is much less overhead in transferring data using `urllib2` than with arizonatransfer and IFTD, because the latter two must identify which protocol(s) to use before they can begin transferring data.

For files less than 10MB in size, IFTD has a higher bandwidth than arizonatransfer. This is partly because arizonatransfer loads and initializes its transfer protocol modules each time an application calls its `getfiles1()` method, and because it receives data to a temporary file which is then moved to the application's requested destination path using the Python `shutil` package. IFTD does neither of these; instead, it loads and initializes its protocols only once when it starts up and writes data chunks directly to the destination path. Consequently it has better performance for smaller files, even though using it incurs the cost of an

XMLRPC round-trip.

When processing files on the order of 10MB, IFTD is noticeably slower than arizonatransfer. Although the data show that IFTD is still slower than arizonatransfer at transferring files on the order of 100MB, the difference in bandwidth for 100MB files is significantly less than it is for 10MB files. The reasons for this are unclear.

Even though `urllib2` had the highest bandwidth in all cases, the data show that IFTD was able to maintain bandwidths comparable to arizonatransfer. This suggests that IFTD may be a suitable replacement for arizonatransfer in the Stork package manager.

## 4.3 Recoverability Performance

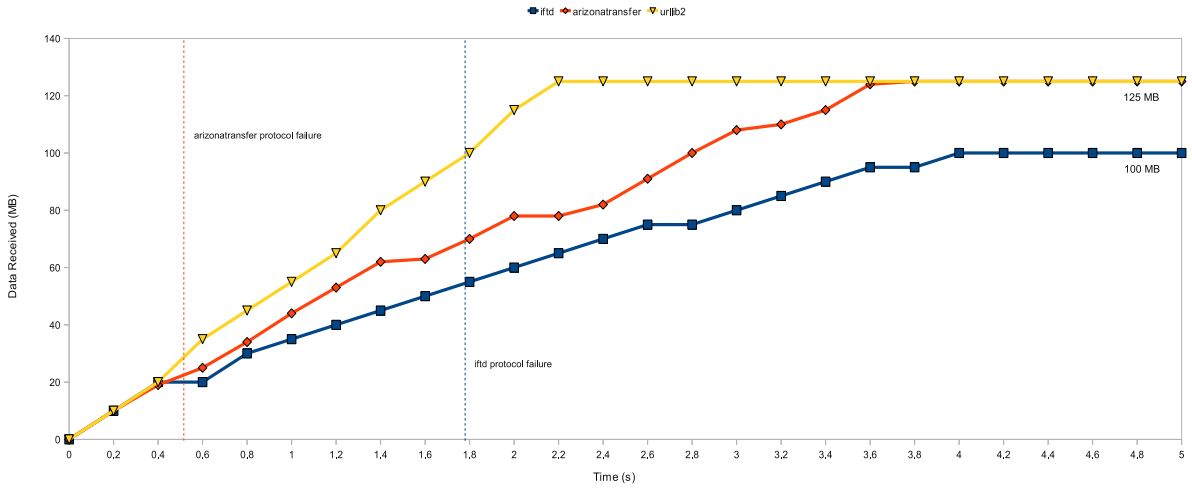


Figure 8: Number of bytes received by each data transfer software as a function of time when a 100 MB file is re-requested after one fourth of it downloads. Although `urllib2` and arizonatransfer have higher bandwidths than IFTD, they transfer more data than IFTD since they are not resumable.

When the unit cost of moving data across a network is nontrivial, data transfer resumability and recoverability become desirable features in a transfer service. The purpose of this experiment is to measure how much data is transferred over time when both arizonatransfer and IFTD use two different HTTP protocol implementations, one of which is programmed to fail after transferring one fourth of the requested data. Both arizonatransfer and IFTD are expected to tolerate this protocol error and continue to receive data. Arizonatransfer is expected to fail over to its next available protocol and in doing so re-transfer the first fourth of the data, whereas IFTD is expected to use its still-operational protocol to receive only the remaining three-fourths of the data. At each 0.2 second interval, the experiment measures how much data arizonatransfer and IFTD were able to download with one faulty HTTP protocol implementation. The total amount of data to transfer is 100 million bytes. IFTD's chunk size was 5 million bytes. The remote host

used CherryPy 3.1.2 to serve data via HTTP on two different ports.

For comparison, the experiment calculated the amount of data transferred by `urllib2` when using `urllib2` to set up a connection to the remote host, transfer 25 million bytes, close the connection, re-open the connection, transfer all 100 million bytes, write the data to disk, and close the connection again. The reason for this particular behavior is to emulate the behavior of a hypothetical application that repeatedly attempts to receive data with `urllib2` by re-requesting the data if the previous transfer request fails. In this scenario, the first transfer request fails after receiving 25 million bytes, and the second request succeeds in receiving all 100 million bytes (meaning that the application transfer 125 million bytes before successfully receiving the data).

The times of arizonatransfer's and IFTD's protocol failures are indicated in Figure 8 with vertical bars. IFTD receives 100 million bytes, whereas both `urllib2` and arizonatransfer receive 125 million as expected.

IFTD only receives 100 million bytes and encounters the error after receiving 50 million bytes because it concurrently uses both its faulty and operational HTTP protocol implementations to receive data in 5 million byte chunks. By the time its faulty HTTP implementation fails, both it and the operational HTTP implementation will have received 25 million bytes each. IFTD uses the operational HTTP implementation to receive the remaining 50 million bytes of the file, despite the failure of the faulty HTTP implementation.

In this experiment, the data show that IFTD's protocol architecture and fault tolerance give it the ability to use DCPs like HTTP to avoid re-transferring data that has already been received from a remote host without IFTD. This resumability allows an application to receive less data than it would have with arizonatransfer or this particular usage of `urllib2` in the event of one or more protocol errors.

## 4.4 Protocol Classification

One benefit of using a classifier to select the best protocol for a data transfer is that it allows IFTD to choose low-latency but low-bandwidth protocols to transfer small amounts of data, and high-latency but high-bandwidth protocols to transfer large amounts of data. In this experiment, IFTD uses two different HTTP implementations to transfer data of various sizes from a remote HTTP server. The first HTTP implementation, `http_slow`, is programmed to receive data as fast as possible for small files, but to introduce a sharp, artificial delay for each chunk proportional to the inverse of the file size when transferring large files. The second HTTP implementation, `http_delay`, is programmed to wait 10 seconds before receiving data and then to receive data as fast as possible. Their bandwidths, calculated as an average of 10 consecutive transfers for each given file, are plotted in Figure 9 on a logarithmic bandwidth scale to emphasize the point where `http_delay`'s bandwidth exceeds `http_slow`'s bandwidth. For the purpose of this experiment, IFTD

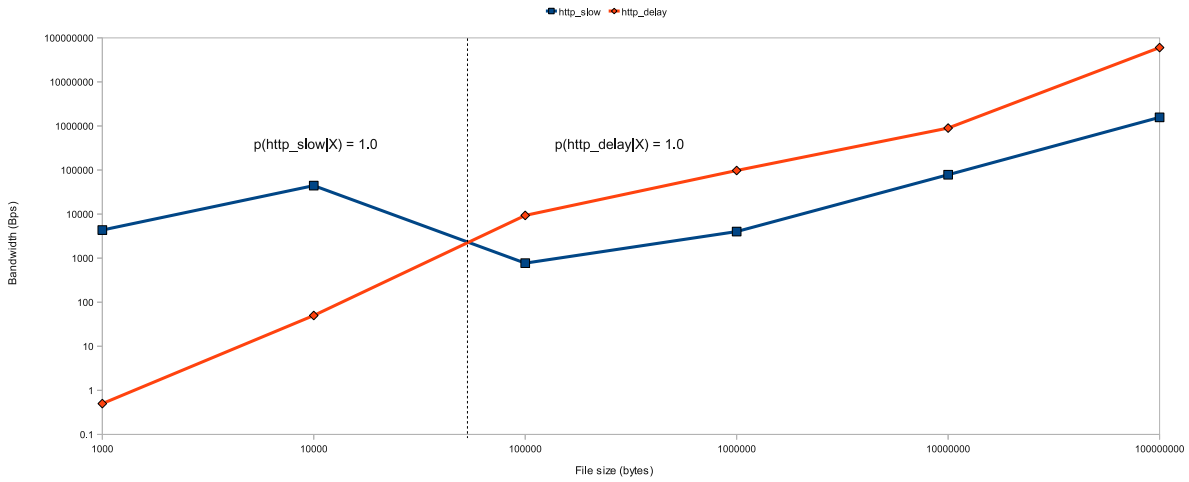


Figure 9: The IFTD-measured bandwidths of two different HTTP implementations for different file sizes. IFTD chooses `http_slow` for smaller files since it has a higher average bandwidth than `http_delay` for small amounts of data. IFTD chooses `http_delay` over `http_slow` once the files become big enough that transferring it with `http_delay` is faster.

is configured to refine its classifier after every 10 transfers.

After the experiment transfers each file 10 times from the remote host, it checks the probabilities of each protocol given the data feature vector. The experiment always chooses a chunk size for IFTD to guarantee that there will be 20 evenly-sized chunks to transfer. The probabilities of each protocol given the data features correlate strongly with the protocol with the highest average bandwidth for that data–in this case, the probability of either protocol given the data feature vector is either 0.0 or 1.0, depending on which received the most data for each set of transfers.

To summarize, in this experiment the data show that IFTD can detect and correctly associate small data transfers with a low-latency, low-bandwidth protocol and large data transfers with a high-latency, high-bandwidth protocol. The probabilities of both protocols being the best for the transfer given the data correlate with the prior bandwidths IFTD had measured while using them.

# 5 Discussion

The experiments presented in this paper are conducted in an idealized environment. In all cases, there is one source host and one destination host connected on a LAN, there is no other network traffic, the bandwidth is practically free, and the system load on both hosts is minimal. This allows IFTD to benefit from using additional but less desirable protocols to transfer data while incurring little cost, it allows IFTD to, in some cases, receive more data than necessary, and it mitigates the advantages of identifying the best data transfer

protocol.

## 5.1 Handling Protocol Behavior

In real-world uses, there is a cost associated with using an active protocol to send or receive. Some protocols may require a nontrivial amount of memory or CPU time to run effectively due to features such as internal bufferring, internal integrity checks, encryption, etc. While IFTD measures protocol bandwidth to decide which protocol is the best protocol for receiving data, it does not consider how using the protocol affects the system's available resources. This is problematic for applications that already require a large percentage of the system's resources, since using IFTD to handle such an application's transfer needs may accidentally cause the system's resources to become too scarce for the application to run effectively. While an application may inform IFTD in advance of which protocols are acceptable to use to mitigate this problem, ideally it would only need to inform IFTD of its maximum tolerances for system resource usages and have IFTD only use protocols that keep its resource usage below the maximum.

Another real-world consideration is that the protocol with the highest bandwidth may not be the most desirable protocol for a particular application. For example, the protocol with the lowest latency may be preferable to protocol with the highest bandwidth in certain cases, and the protocol that can compress data to the smallest possible size may be preferred in other cases. One future improvement on IFTD is to allow an application to identify a set of features for IFTD to measure in order to cause the classifier to favor protocols based on different criteria than the highest bandwidth.

The fact that IFTD starts passive receivers automatically, regardless of whether or not they have historically been the best protocol with which to receive data, poses a problem for applications in situations where bandwidth is expensive. The experimental results from Figure 2 indicate that using an NCP concurrently with one or more DCPs can increase the number of duplicate chunk transfers over 50 percent. However, as seen in Figure 6, there are scenarios where an NCP is considered to be the best protocol with which to perform transfers since it has the highest bandwidth. Also, using a DCP while an NCP is *not* receiving data, like how the HTTP protocol received all 65 of its chunks before the BitTorrent protocol began to receive, could potentially increase IFTD's total bandwidth without using too many system resources. These observations suggest that the best protocol with which to receive data can change during the course of a data transfer, and that the calculation for the best protocol must consider more information than average bandwidth.

One possible improvement to IFTD's data transfer algorithm is to recalculate $p(X_{now}|b)$ periodically during the data transfer and then calculate the best protocol $b$ given data feature vector $X_{now}$ representing the data transferred thus far. If $b$ changes, and IFTD is receiving data from a remote IFTD, it would inform

the remote IFTD of the change. However, communicating with the remote IFTD would incur the cost of an XMLRPC round-trip, and it is not immediately clear how representative $X_{now}$ is of the data yet to be transferred. Also, the relevance of prior completed transfers should be considered, since the current transfer may be unrepresentative of typical data transfers for this classification of data. The weight of each calculation used to choose protocols that maximize data transfer bandwidth still need to be explored.

Another possible improvement is to make use of the fact that data used by the naive Bayes classifier can be used to calculate $p(b_i|X)$ for any protocol. As a result, IFTD can calculate the best $k$ protocols to use to transfer data with features $X$, where $k$ can be given by the application. Doing this means that IFTD does not necessarily need to start all passive receivers; instead it may have at most $k$ receivers running at once regardless of whether or not they are passive or active. This represents a compromise between running all receivers, which potentially takes up too many system resources, and running only the best receiver, which increases the likelihood that a source host sends the destination host data only to have it rejected since the data cannot be received by the currently running protocol. The effects that the values for $p(b_i|X)$ for the $k$ best protocols and the effects that the choice for $k$ given $n$ protocols have on the choice of the best protocol are yet to be determined.

## 5.2 Choosing the Best Protocol

There are many alternative methods of calculating the best protocol with which to transfer data given IFTD's architecture. For example, in order to consider the system-wide effects of using a given protocol, IFTD could record the average $bandwidth/(1 + \Delta CPU)$ or $bandwidth/(1 + \Delta Memory)$ or a combination of the two, calculated per transfer, in order favor high-bandwidth protocols that increase IFTD's CPU and memory usage minimally. Also, IFTD could instead calculate $1/latency$ for each protocol to favor the protocol with the lowest latency. Additionally, an application could supply IFTD with weights for each protocol's $p(b_i|X)$ based on application-defined desirable traits, such as the presence of data encryption or the usage of certain port ranges. This favorability information may be beyond the scope of IFTD to calculate but may affect the usability of the protocols enough that providing it is warranted for a specific transfer. Future experimentation with IFTD may determine which if any of these alternatives allow IFTD to make better decisions in general.

Since IFTD depends on a naive Bayes classifier for choosing the best protocol, the choices for the features it measures for data transferred were made with the intuition that they have little or no correlation among themselves in general. If this is true, then the assumption made by the classifier that $x_i$ is independent of $x_j$ when $i \neq j$ is well-founded. However, a naive Bayes classifier can perform well even when its probability assumptions are incorrect [16]. Consequentially, IFTD may make more accurate choices if it considered some

(but not many) additional features. For example, IFTD could record the set of IP addresses from the remote hosts it engages during data transfer, the port number(s) used during the transfer, and some information about the remote services engaged to perform the transfer (e.g. the version number, software brand name, etc.), since these features may also have an effect on how well the data is transferred. Further experimentation with real-world data transfers is needed to determine which subset of features are the most useful for choosing the best protocol.

# 6 Conclusion

This paper presented the design and implementation of a prototype data transfer service that decouples an application from data transmission. The prototype implementation IFTD provides a protocol architecture that allows for arbitrarily complex data transmission schemes and automatically makes every protocol it uses resumable. It also remembers each data transfer it performs so it can intelligently choose which protocols to use based not only on the capabilities of the remote hosts it engages, but also based on the properties of the data itself. Additionally, IFTD can use existing data transfer services to perform a data transfer, allowing it to be incrementally deployed. Applications using IFTD do not need to implement protocol-specific content negotiation, data transmission, or error handling logic in order to perform data transfers.

This paper also presented a set of four simple performance evaluations of IFTD. While using IFTD adds transfer overhead, its automatic protocol resumability allows it to avoid re-transferring data it has already received, as well as tolerate multiple fatal protocol errors. Additionally, it has been shown to correctly identify the protocol with the highest bandwidth over time with successive data transfers. While the evaluations were limited in scope, the results indicate that pursuing further development and experimentation with IFTD is a worthwhile endeavor.

# 7 Acknowledgments

The authors wish to thank Scott Baker for his continuing feedback on this project, the University of Arizona for supplying the development and evaluation machinery, the numerous developers of Stork for arizonatransfer, and the NLTK team for their statistical analysis code. Additionally, Mr. Nelson wishes to thank Dr. Hartman for guiding him through the development of IFTD.

# References

[1] Application layer protocol examples. `http://en.wikipedia.org/wiki/Application_Layer`.

[2] BitTorrent Protocol. http://www.bittorrent.com.

[3] Cappos, J., Baker, S., Plichta, J., Nyugen, D., Hardies, J., Borgard, M., Johnston, J., and Hartman, J. H. Stork: Package Management for Distributed VM Environments. In *Proc. 21st Systems Administration Conference (LISA '07)* (Dallas, TX, Nov 2007), pp. 79–94.

[4] Cappos, J., Samuel, J., Baker, S., and Hartman, J. H. Package management security. Tech. Rep. 08-02, University of Arizona, 2008.

[5] CherryPy: a pythonic, object-oriented HTTP framework. http://www.cherrypy.org.

[6] Cisco Systems. Cisco Visual Networking Index: Usage Study. White paper, Cisco Systems, Inc., Oct 2009.

[7] Debian: The Universal Operating System. http://www.debian.org.

[8] Fedora Project. http://fedoraproject.org.

[9] giFT: Internet File Transfer. http://gift.sourceforge.net/.

[10] Hüni, H., Johnson, R., and Engel, R. A Framework for Network Protocol Software. In *ACM SIGPLAN Notices* (Oct 1995), pp. 358–369.

[11] Jigsaw Download. http://atterer.net/jigdo/.

[12] Microsoft IIS. *Bit Rate Throttling*, 2010. http://www.iis.net/download/bitratethrottling.

[13] Norberg, A., Jonsson, M., Wallin, D., and Nelson, C. BitTorrent Implementation. http://www.rasterbar.com/products/libtorrent/index.html.

[14] Planet-Lab. http://www.planet-lab.org.

[15] Python. *Initialization, Finalization, and Threads*, 2.6.5 ed., 2010. http://docs.python.org/c-api/init.html.

[16] Rish, I. An empirical study of the naive Bayes classifier. Research report, IBM, Yorktown Heights, NY, Nov 2001.

[17] Rodrigues, R., Castro, M., and Liskov, B. BASE: Using Abstraction to Improve Fault Tolerance. In *Proc. 18th SOSP* (Banff, Alberta, Canada, Oct 2001), pp. 15—28.

[18] Sherwood, R., Braud, R., and Bhattacharjee, B. Slurpie: A Cooperative Bulk Data Transfer Protocol. In *Proc. INFOCOM 2004* (Hong Kong, Mar 2004).

[19] St.Laurent, S., Johnston, J., Dumbill, E., and Winer, D. *Programming Web Services with XML-RPC*. O'Reilly Media. O'Reilly & Associates, Inc., Sebastopol, CA, 2001, pp. 1–32.

[20] Tolia, N., Kaminsky, M., Andersen, D. G., and Patil, S. An Architecture for Internet Data Transfer. In *Proc. 3rd NSDI* (San Jose, CA, May 2006).

[21] Ubuntu. http://www.ubuntu.com.

[22] urllib2 source code. http://code.reddit.com/docs/urllib2-pysrc.html.