DTS: The NOAO Data Transport System

Michael J. Fitzpatrick*
National Optical Astronomy Observatory, 950 N Cherry Ave, Tucson, AZ 85719

ABSTRACT

The Data Transport System (DTS) provides automated, reliable, high-throughput data transfer between the telescopes, archives and pipeline processing systems used by the NOAO centers in the Northern and Southern hemispheres. DTS is implemented using an XML-RPC[1] architecture to eliminate the need for persistent network connections between the sites, allowing each site to provide or consume services within the network only as needed. This architecture also permits remote control and monitoring of each site, and for language-independent client applications (e.g. a web interface to display transfer status or a compiled task to queue data for transport which is more tightly coupled with the acquisition system being used). The resulting system is a highly multi-threaded distributed application able to span a wide range of network environments and operational uses.

Keywords: Data transport, observatory operations, software architecture, middleware

1. INTRODUCTION

The DTS project was initiated specifically to meet the data transport needs of the Dark Energy Survey (DES) scheduled to begin science operations at the Cerro Tololo Inter-American Observatory (CTIO) Blanco 4m telescope in 2011. The Dark Energy Camera (DECam) used by the survey is a 520 Megapixel mosaic of 62 2k x 4k imaging CCDs, producing roughly 500-600 MB per exposure (compressed). The DES observing cadence is expected to generate some 200-300 GB/night that must be moved to the DES pipeline facility at the National Center for Supercomputing Applications (NCSA) within 18 hrs of the start of observing. When not being used for the Dark Energy Survey, the instrument will be available for community use on the Blanco 4m where the observing cadence and data rates will vary by program. All raw data will be archived at the NOAO/Tucson archive facility, however the data paths will differ depending on whether a particular image is acquired part of the DES. Additionally, a second level of differentiation is required for DES data due to various sub-surveys being carried out concurrently; these data may have another set of data path requirements.

Although begun to meet the needs of a specific project, the DTS is also intended to replace the existing data-transport infrastructure currently in use at NOAO[2]. While the basics of data transport and routing will be the same, the instrumentation used will vary considerably. This means the system has to be configurable for a wide range of data sizes and observing modes; an optimal transfer method for a single 500MB image may not be ideal for many smaller files such as spectra, and the constraints on when they must be available call for added flexibility in tuning aspects of particular data paths.

Meeting the timing requirement of DES data transport (i.e. 18-hours from the start of observing) is a challenge because of the limited bandwidth currently available from the CTIO facilities. The initial transport method implemented uses a parallel-socket scheme that makes optimal use of the network, and DTS is extensible to use other transport models if needed. Lessons learned from our current data transport software, both in terms of its implementation as well as the operational realities, strongly guided the DTS design presented here.

Even though we can expect the overall bandwidth and connectivity to be improved prior to the start of observing with DECam, the DTS must meet requirements beyond just bulk transport. A majority of the development effort has been directed towards building a highly flexible system to meet all of the operational modes in which we expect to eventually use the system. Indeed, several new use-cases have been identified since the start of the implementation phase of the project and we expect to use some of these as test beds to stress the system and prove its reliability prior to operational deployment for DES or within other NOAO mission-critical observing environments.

*fitz@noao.edu; phone 1 520 318-8387; fax 1 520 318-8460

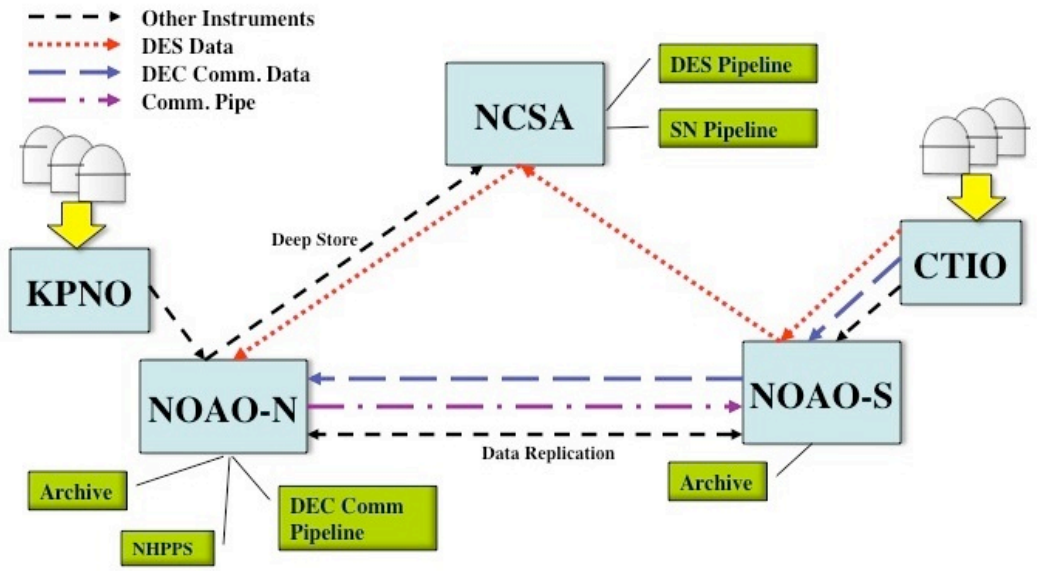

Figure 1. A diagram of the DTS data flow once it is fully deployed. As data make there way through the system, different actions are required at each stop (e.g. archive ingest, pipeline delivery etc) before moving on to the next queue. The DTS uses external processes to manage these "delivery" actions rather than implicit knowledge of what is required at each location.

1.1 Overview of DTS

The DTS uses an XML-RPC architecture to provide a suite of basic services at each site without requiring persistent connections within the system. The central component is a DTS Daemon (**dtsd**) process responsible for managing the data queues and responding to client service requests (which, in most cases, will be other **dtsd** tasks in the system). The **dtsd** can be configured to run as a standard Unix service, providing continuous availability and automated recovery in the event of a system crash. The **dtsd** is a highly multi-threaded application capable of managing any number of transport queues and client connections. Because of its service architecture, the DTS can be easily monitored or managed by remote clients, simplifying operations of the distributed system from a central location.

Data are introduced to the DTS system via a specialized client called **dtsq** who is responsible for moving the data from the local machine to the (possibly remote) machine hosting the DTS. In the observing environment, a **dtsq** would be triggered at the end of each exposure, moving the data from each telescope dome to a centralized 'mountain cache' where the DTS would then queue the data for transport to a downtown facility and beyond. This replaces the current '*lpd*' based scheme, but still provides the same quick response and recoverable queuing functionality.

A simple **dtsmon** application is available and runs on a reserved port, allowing all system messages to be centrally monitored. A command-line tool called the **dtsh** provides a means to test system services as well as a scripting capability to automate actions (e.g. generate transfer reports, manage queues, etc).

Transfer queues may be configured individually and operate as a *normal* queue (FIFO fashion), as a *scheduled* queue (triggered at fixed intervals, e.g. a specialized queue to transfer daily logs), or as a *priority* queue for time-sensitive delivery. Priority queues will block other transfers until empty to make all the network resources available for transport and won't normally be configured for regular use.

While the DTS can be configured as a system of federated sites and transport queues, it can also be used for personal use by simply starting a **dtsd** on a remote machine and using the **dtsh** application to transfer a file. Because the system provides basic filesystem services, recursive transfers of directories and remote updates similar to *rsync* are also possible.

2. ARCHITECTURE

The Data Transport System, by definition, has to run on multiple machines at multiple locations; to further complicate matters, the activities required of DTS at each site will differ. This led us to consider a service-oriented architecture as the most appropriate basis for the solution. We wanted to avoid requiring persistent connections between sites since this greatly complicates the error recovery in the event of a system crash or network outage (e.g. state information must be restored, socket connections need to be re-established, etc), and because we were essentially designing a distributed application, a Remote Procedure Call infrastructure would satisfy our needs.

XML-RPC provides a robust and highly capable means of implementing the services we require. It is also relatively lightweight in the sense that remote method calls do not need to exchange a lot of information in order to succeed and HTTP transport is well established and reliable. The DTS can then be designed in terms of how data must flow through a system of services available at each site instead of a more monolithic approach using a client/server or master/slave design pattern. This leads to a decentralized system that also satisfies our desire to operate and maintain the system from more than one location. The implementation is a specific service is also simplified by allowing us to concentrate on the functionality the service provides rather than on how it might be used.

2.1 Bulk Data Transport

We realized early in the design that given our operational requirements we would need to separate the command-and-control parts of the DTS system (e.g. the remote monitoring or functional aspects of managing the transport queues) from the file transfer itself (i.e. a failed file transport should not be capable of bringing down the entire system). This design point allows us to think in terms of performing the bulk data transport independently, and since we would need to manage multiple transport queues concurrently anyway, it provides a natural split of the task execution onto a separately running thread where the only requirement is to transport the data object. The actual protocol used is then irrelevant to the rest of the DTS and can then be replaced in the future if something better is found, or we can provide multiple transport protocols that can be configured for a particular data path or destination.

The default transport method implemented in DTS utilizes a parallel set of TCP/IP socket connections to *stripe* a data object for transport to the destination machine. This means that e.g. over 10 threads of execution each transport thread is responsible for sending a different 10% of the total file, and if each thread can only effectively utilize only 10% of the nominal bandwidth (e.g. due to network routing issues) we can effectively get a 100% utilization of the network. In reality, we won't actually see this level of network efficiency or fine-grained control, however the concept of parallel-transport provides a means to maximize the network efficiency we have available. The application can be configured to use the number of threads required to meet the specifications, or can be tuned to use a different number of transport threads based on operational experience with a specific network connection. Low-bandwidth or high-latency network paths might require a larger number of transport threads to work as needed, *"local"* transport connections might meet the same throughput values with fewer *stripes*, but DTS is able to manage these connections independently.

2.2 Configurability

One key design element of the DTS system is the ability to configure it *as a system* for deployment in a completely new scenario. Each DTS instance at a given site in the network will have different requirements; some will only be ingesting data for transport, some will be endpoints responsible for delivering data, and others will have a mix of responsibilities. These responsibilities are determined by the data queues being operated at a specific site, each of those queues may in turn be doing something different depending on where it is in the data route.

A simple text file of keyword/value pairs is used to define the needed parameters of DTS at a given site (i.e. it's name, host machine/port, etc) as well as each instance of a data queue to be managed on the machine. The queue configuration block likewise defines the parameters controlling the behavior of that queue (i.e. the ingest/delivery application to use, where to put the data, the next leg of the transport route, etc). Default values are used for those parameters not explicitly defined. The configuration file may be created individually for each site, or a *master* file configuration file can be written to define all nodes in the DTS network (on startup, DTS applications requiring the configuration file will use the host name to find the appropriate configuration block appropriate for that machine). This latter method has the advantage that each DTS site is able to know which other DTS sites are available (e.g. a DTS application might periodically *ping* all DTS sites in the network to ensure their availability). RPC methods exist to allow this master file to

be broadcast to all DTS sites; a *restart* command can then be issued to use the new configuration parameters. This allows a central operator to manage the DTS system with minimal effort in response to changing network conditions or observing load.

2.3 Sandbox Security

The whole point of the DTS is to move data from one machine to another, however exposing a network service designed to give remote machines read/write access to your disks is inherently risky. For this reason, DTS implements a sandboxed view of the filesystem so that only directories or files below the designated DTS root are visible to the RPC services. In cases where a pathname is needed as an argument for the RPC method, the implementation of that method in the server passes the argument through a filter that strips it of directory elements (e.g. the *"../"* directory entry) that might allow access outside of the DTS root. Running a DTS server at a given site can then be done such that the DTS root can be limited to a specific disk partition in order to limit its maximum size, and sites are free to designate whichever root directory is appropriate for their needs.

In cases where an RPC method provides a more general file operation (e.g. to remove a file, create a directory, rename a file, etc), the server-side methods require a password argument that is validated against the one in the configuration file before processing the request. We assume that a trusted client would know the password, either because we share a common configuration file or we've otherwise agreed to share the password and expose our disk resources. If more security is required it is possible to require the password for all RPC methods, however this would be a future enhancement of the system.

2.4 Implementation

The DTS is implemented entirely in portable ANSI-C code. The XMLRPC-C[3] open-source library, using the cURL[4] library to provide the underlying HTTP transport mechanism, is used to implement the low-level XML-RPC protocol. A simplified abstract interface to XMLRPC-C (*libxrpc.a*) was written not only as a convenience, but to isolate the low-level interfaces from the DTS library and applications layers in the event a different implementation of XML-RPC was ever required. A DTS library (*libdts.a*) providing client and server-side code, and the re-usable functionality needed by all applications (e.g. starting the server, adding a callback method, calling a remote method, etc) is layered on these libraries. DTS applications are then built entirely on the DTS library with no implicit knowledge of the underlying infrastructure.

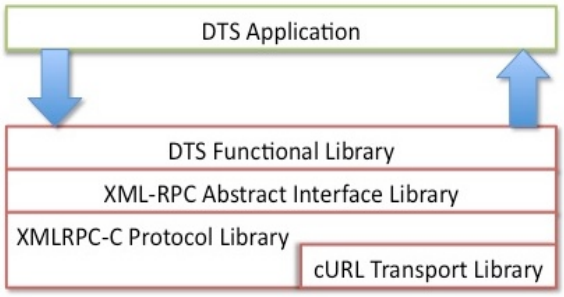

Figure 2. Architecture of a DTS application.

Documentation of the libraries and applications is generated automatically using the *Doxygen*[5] application and coding techniques. Unix-like *man* pages are written for each application to describe their options and functional capabilities, a *User's Guide* for the entire DTS system is in preparation.

Excluding the third-party XML-RPC/cURL library code and Doxygen-generated documentation, DTS represents some 30K lines of code to implement the functioning system, however the underlying DTS library comprises roughly 85% of the DTS system code base and so we expect additional applications development will require minimal effort to extend the system for new or alternate capabilities.

3. SYSTEM COMPONENTS

As we saw earlier, DTS applications are layered upon two libraries that isolate the applications from the details of the XML-RPC protocol implementation. Although XML-RPC is a widely used standard and there are plenty of implementations available, the choices for a particular development language are much more limited. We chose XMLRPC-C as the best of the C implementations evaluated, but also decided that a simpler interface was required since we would only be using a small fraction of the functionality available in the protocol. This became the foundation for

the main DTS system library written to implement the needed client and server-side capabilities used by the system applications.

The initial DTS release requires only four applications: The DTS Daemon (**dtsd**) is the primary application running at each site, providing basic DTS functionality and data queue management; the DTS Queuing Agent (**dtsq**) is responsible for submitting a data object to the DTS system for transport; the DTS Shell (**dtsh**) utility can be used interactively to interact with the DTS system or in batch mode to automate DTS actions (e.g. generate daily status reports); The DTS Monitor (**dtsmon**) is a utility used to monitor activity of the DTS system.

3.1 DTSD: The DTS Daemon

The DTS daemon is the one application required for all sites within the DTS network. The DTSD provides all of the XML-RPC services currently implemented and is also responsible for the management of transfer queues. It supports both synchronous and asynchronous method calls and can service multiple clients simultaneously. Typically method calls happen between DTSD applications running at each site, e.g. as part of the calling sequence necessary to negotiate a transfer. Other DTS applications however will communicate with the DTSD to submit new data for transport, or from other clients used for maintenance and monitoring of the system.

On startup the DTSD will initialize the root directory specified in the configuration file. All needed files and directories will be created and initialized, if a previous DTSD was running and an error occurred then any necessary cleanup will be done. The RPC methods are then installed in the server and threads are spawned to manage each of the queues operating at that site. DTSD may be run as an interactive application and will accept a limited number of commands from the user, the default action however is for the task to fork itself and continue running in a child process in the background. For our use, DTSD will be configured to run as a Unix system daemon so its services can be started automatically at times when it isn't already running. This has the extra advantage that in the event of a hardware failure the system will restart automatically the next time a remote DTS tries to connect.

The DTSD application, and therefore the entire DTS system, is designed to be highly robust and we do not expect to have to restart it often. Our system architecture supplies the DTS library used by the applications, and since this library provides both the client-side code as well as the server method implementation we can be confident the RPC (and any needed error recovery) will work as we expect.

Most of the behavior of the DTSD can be controlled by parameters defined in the configuration file. The XML-RPC server in the DTSD requires a single TCP/IP port number to visible through any external network firewalls to allow RPC methods to be called.

3.2 DTSQ: The DTS Queuing Agent

The DTSQ application is responsible for submitting a data file to the DTS system for transport; it acts similarly to the way the *lpr* command functions in the current *lpd*-based transport system[2]. DTSQ is normally triggered at the end of an exposure by the image acquisition system automatically to provide a pristine copy of the raw data for archiving and prior to the observer receiving a copy of the image for their use. Since DTSQ is invoked as part of the data-taking system, one critical requirement is that it returns control to the calling system as soon as possible to avoid delaying the next exposure. This is accomplished by forking off the process to do all of the expensive processing once the task has determined that it is able to contact the target DTS daemon and that the DTSD will accept the file. Although computing checksums, transferring the file to the common mountain DTS system and completing the transaction may take several seconds or more, control is normally returned in fractions of a second and the caller knows immediately whether an error has occurred or if the system is proceeding normally.

The DTS file transfer mechanism is implemented as a series of XML-RPC calls between hosts and so DTSQ contains a small XML-RPC server to provide the minimal set of server methods needed for the transfer. In the expected deployment, a central DTS system will collect data from multiple sources on the mountain, each DTSQ will provide the transfer sockets used so the single DTSD can make outgoing client connections rather than manage a pool of server socket connections itself.

The DTSQ uses a configuration file to minimize the number of command-line arguments required when invoked. This file typically contains definitions for each of the queues to be used and provides a means to tie a simple queue name to details about the DTSD host and preferences for the transfer. The DTSQ is then started with a command such as:

```
dtsq –q mosaic /home/data/image0312.fits
```

where the *–q <name>* argument gives the name of the queue to which we're submitting. When the command starts, the DTSD is contacted to ensure it is running and the network connection is working, the task then checks that there is sufficient disk space available for the file and that the queue is active; If any of these tests fail the DTSQ returns with an ERR condition, otherwise the task forks off to continue processing and the parent returns an OK status. In the forked process, the MD5 sum of the file is computed and the transaction with the DTSD is completed. If at any point the transfer fails, the filename is written to a *recovery file* that can later be used to re-queue each of the images in the original sequence. Recovering in this way requires intervention by an operator at some point, however observing is not affected. On successful submission to the queue, the task leaves a *token file* in the DTSQ configuration directory (typically $HOME/.dtsq), creating the needed directory tree structure so as to mirror the path to the submitted file. This token file contains details about the transfer (time completed, status codes, etc) and by retaining the directory structure we're able to easily match our token files to real files on the machine for possible cleanup or recovery.

3.3 DTSH: An Interactive Shell

The DTSH serves primarily as an interactive DTS shell during development and can be used to manually issue commands to a DTSD. This is convenient when implementing a new server method since it allows a direct means to invoke the method from a client. Because all DTSD services are accessible, it may also be used as a command-line tool for manually transferring files or performing maintenance on a remote DTS site (e.g. to halt/restart a queue, check on available disk space, etc).

As with other Unix shells, DTSH also provides a scripting capability to automate repetitive tasks or to build utility commands for operational use. For example, simple utility scripts might look something like:

```
#!/usr/local/bin/dtsh –f
ping siteA ; ping siteB
sleep 60
rewind
```

```
#!/usr/local/bin/dtsh –f
push $1 siteA:/eng/nite1 &
give $1 siteB:/data/
```

On the left is a *heartbeat* script that *pings* two of the DTS sites, then uses the built-in *sleep* and *rewind* commands as a primitive loop before repeating; an operator could use this to monitor the health of the system or a more complex script could be used to feed a web page with status monitoring information. On the right is a utility for transferring data to two different sites simultaneously that uses the built-in '*&*' operator to run the first command on a separate thread before beginning execution of the second command using a different transfer protocol. The *'$1'* operator represents the filename argument given to the script. Other functionality such as output redirection or use of shell-escapes is not shown here however a full list of capabilities and examples are provided in the manual page for the task and other DTS documentation.

3.4 DTSMON: The DTS Monitoring Application

The DTSMON utility is currently implemented a text-based application that listens to messages from the DTS system on a TCP/IP socket on whichever machine is being used. Once started, it supersedes any other monitor currently running (although we are considering a mode to allow multiple monitors) and need not be run from a machine currently in the DTS network, allowing monitoring by operations personnel from anywhere in the world. Eventually, a GUI will be added to permit filtering of message streams to look at a specific site (something which can currently be done with a *grep* filter of the text output) and if security becomes a concern it can be password-enabled if needed.

All DTS applications will attempt to connect to the DTSMON when they first start up and key parts of the system code use an internal *dtsLog()* procedure to send information to the monitor (e.g. begin transfer, failed DTS transfer, etc) if a running DTSMON is detected. The DTSMON will "announce" itself to the DTS on startup allowing other applications to then connect and resume logging of messages.

4. BULK DATA TRANSPORT

The transfer of a file between two machines in the DTS is only conceptually a single operation. Using the parallel socket method currently implemented, a file is divided into *stripes* where multiple threads are responsible for

transferring only a fraction of the file. Within the transfer of each stripe, data are written in *chunks* where the chunk size can be tuned to be optimal for the TCP/IP window appropriate for the network connection being used. At the lowest level, each chunk is transferred using multiple calls to the system *read()/write()* procedures, since these do not guarantee that a single call will transfer the requested number of bytes, repeated calls are made until the complete chunk is transferred. This can result large numbers of I/O operations being performed if the chunk size is not properly tuned, but because transfers are parallelized over multiples stripes we are able to maximize the compute resources doing the transfer and minimize any inefficiencies found along a particular network transfer path.

TCP/IP window tuning has long been a common practice among network system administrators to improve performance. In a system such as DTS where individual data file sizes might vary widely, these tuning parameters may in fact degrade transport performance for some instrumental data. We assume the *stripe* and *chunk* sizes being used by DTS are based on operational experience with the system and provide the best overall performance of the connection given real-world requirements for efficiency. Recent Linux kernels provide auto-tuning of the TCP/IP window sizing (and we expect this to improve and be more common in the future), so we've chosen to avoid this level of fine-grained control of the network connection in the bulk transport code.

4.1 Transport Methods

The *method* of transport refers to the low-level protocol used to transfer files between DTS hosts. At present only the parallel TCP/IP socket scheme is supported (in any case, the *default* method is always configured as using the *dts* transport method). Because the transport is isolated from the mechanics and negotiations required in the DTS upper levels, we are free to use alternate methods to actually send the data to a remote system. This may in principle include GridFTP[6] or UDP[7] protocols such as FDT[8], or common utility commands such as *scp* or *rsync*.

The realities of implementing an alternate transport method are more complex, however. In the custom parallel method we've implemented we were able to use the model of a single file accessed by multiple threads to stripe the data, implementing a method such as GridFTP with it's own model of parallel transfers means the changes required to support that method are more extensive. Fortunately, we have no compelling reason thus far to support alternate methods but we remain open to the idea of implementing other methods in the future.

4.2 Transport Modes

The *mode* of transport can be defined in the configuration of a queue as either **push** or **give**. In a *push* model, we assume we are contacting a DTS service willing to provide the resources needed to accomplish the transfer, i.e. that it will create transfer sockets we can use and will assume responsibility that those are unique resources for the request. In this mode, a sending machine makes a request for a *push* transfer and in the reply is informed of which sockets are to be used (or a denial of the request), it then connects and begins transferring the data over the number of threads configured for that connection. In a *give* model, the sender is willing to provide the connection resources (i.e. the transfer sockets) and requests of the receiver that it accepts the file for transport. If accepted, the receiver (i.e. the remote machine in this case) connects to the sockets as a client and the sending machine begins transferring the data.

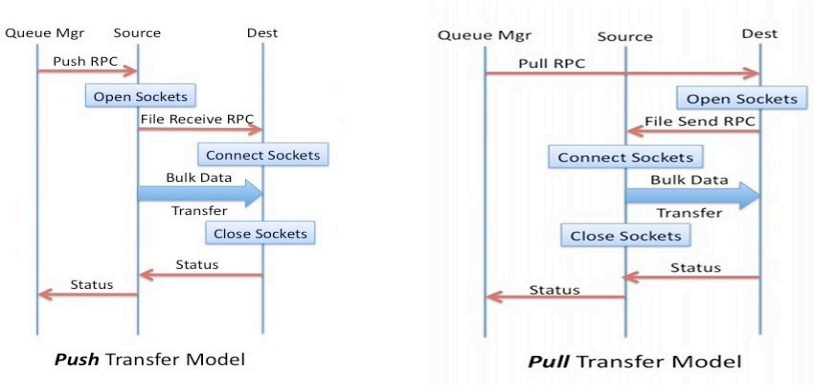

Figure 3. Activity diagram showing the *push* and *pull* transfer models. Multiple transfer modes exist to allow configurable options to manage socket connections. Because transfers are performed using RPC calls to the DTS servers, third-party transfers (e.g. a transfer between two remote DTS sites commanded by an operator locally) are also possible.

Complementary modes also exist in the form of **pull** and **take**, these are similar in the direction of transfer for data but differ in how the RPC methods are used. In a *pull* model we assume we are contacting a DTS service willing to provide resources for the transfer but we act as a third-party and ask the recipient to be the active party in performing the transfer. In a *take* model we likewise send the request to the destination host but ask that it be the passive partner in the transfer, as the sender we've opened the transport sockets and simply want the receiver to connect and accept the data.

In all transfer modes there is a negotiation between the sender/receiver that validates the DTS connection and the receiver's ability to accept the file (e.g. that there is sufficient disk space, the queue is active and accepting new files, and so on). This is true from applications such as the DTSQ as well as for intra-DTS movement of files during queue processing. If a transfer request fails for any reason, data simply back up in the upstream queue until a re-send is attempted.

4.3 Data Routing

DTS sites operating a specific named queue define the routing of data through the system; we assume the configuration file has defined the queues at each site such that the *src* and *dest* parameters for form a contiguous path through the system. Data queues flow in only one direction through any number of DTS sites and data may (in principle) be submitted to any site along the path. The *push* or *pull* behavior of moving data between any two sites, as well as the transport parameters (number of threads to use, delivery applications, etc) can all be defined individually in the DTS configuration file. This allows us to optimally tune each leg of the total path according to its the network throughput, firewall restrictions, method of transfer and so on.

When data are submitted to a specific queue they enter a *spool* directory to await processing much as printer jobs sit in a spool queue. The location of this spool directory, and therefore the destination filename of a given transfer, is negotiated as part of the RPC exchange that takes place when submitting data or transferring files between queues. Because the destination of a transfer is that queue's spool directory on the next machine, normal queue processing at the destination will ensure data continue on the route until an *endpoint* site is reached. Following a successful transfer the receiving site can instruct the sender to free the spool file in order to free up disk space, but until we gain more practical experience with the system we plan to have operations personnel periodically prune the spool directories once the transfers have been verified.

The destination of a particular data leg may be specified in the form of multiple queues to copy the data to a second queue once it has arrived. Transport is done as described above, but before completing the transaction the receiver submits the data to the other named queue (places a copy in a the new spool directory) for processing. This allows us to define a single transfer of data over bandwidth-limited connections (e.g. from the observatory or via international networks), but then split the transfers into multiple paths for transport to collaborating institutions when we have more network bandwidth available. Should a queue then ever need to re-join the original queue or move to a new path entirely, the destination may also be specified as a particular queue on the receiving host.

The *delivery order* of data on queue is guaranteed only from the perspective of the data submitted to the *ingest* site, i.e. although the queue-hopping and arbitrary-submission-site behavior is allowed by the software, using this functionality might result in extra data being seen at an arbitrary receiver compared to what the ingest site might report sending. If the order of delivery is important, the configuration of alternate queues should be considered.

If at any point a transfer along the route fails (e.g. a network outage or the destination machine has crashed) data will continue to flow up to the point of blockage. If *that* machine is then unable to accept new files (e.g. because the DTS disk being used is now full) the upstream site then stops sending data. This practice continues until the backup affects the data submission site where an ERR is returned to the originating DTSQ call and recovery of failed transports is required by the operations personnel. When an error is detected in the DTS queue processing, periodic attempts are automatically made to re-establish contact until normal data flow can occur, the system can also be configured to notify an operator via email that an error requires attention.

4.4 Data Neutrality

To create a generalized transport system and not strictly and astronomical one, we decided to impose a requirement on ourselves that the DTS not be limited to the handling of FITS images only. Indeed, the contents of a file are completely irrelevant to the transport itself; three separate checksum values (MD5, SYSV and BSD) and file status information are used to guarantee the integrity of the file while in the system. Because we implement common filesystem features such as *mkdir*, DTS is able to accept a *directory* for transport and will recursively transport the contents of a directory

automatically if that is what was submitted to the DTS. The DTSQ application also has an option to *bundle* a directory or multiple files into a single tar file for transport, however we rely on the receiving application to unpack the tar bundle if desired.

4.5 Data Integrity

The highest priority for DTS is that it guarantee the integrity of the data being transported and that each file delivered is identical to the one originally accepted for ingest by DTS. To meet this requirement, several layers of data validation are implemented by the system.

At the lowest level of bulk data transport, there are multiple options to checksum the data and validate the transport: each I/O operation may compute a checksum and re-transmit data if there is an error, however this requires a round-trip validation to the receiver for each packet (which may be small) and is highly inefficient over hi-latency connections. A second option is to checksum the *chunk* transmitted, meaning there are generally fewer round-trip messages required but which still impose a delay in the transport. We can next checksum an individual *stripe* of data and if a failure occurs we need to only resend the affected stripe and not the entire file, however in the current implementation of the parallel transfer this failure would block completion of the transfer for as long as it takes to resend the stripe (which is still much less than a resend of the entire file). At the highest level, bulk transport can be configured to checksum the entire file. A failed checksum test will require a retransmission of data (by the designated chunk/stripe/file method) and multiple failures (e.g. a prolonged network outage) will eventually halt the transport and the queue in an error condition. The method of validation used is a configuration file option to allow the best method to be chosen based on the network properties; by default transfers are validated for each stripe.

Above the bulk transport, the DTS performs its own checksum validation of the file once the transport is complete. The checksum values sent in the RPC message (and which exist in the queue spool control file) are recomputed to ensure the received file has the same values. A transfer is not considered valid until the bulk transport returns successfully (which necessarily requires that the *stripe/chunk/packet* validation has also succeeded) and that the DTS queue manager also validates the completed file checksum.

Various checksum values are used at different levels: MD5 is fairly expensive and so is reserved for the final file checksum at the DTS queue level, however BSD/SYSV values are relatively cheap for stripes/chunks and are the values used during bulk transport to validate transfer.

5. TRANSPORT QUEUE MANAGEMENT

Queues are managed by the DTSD application running at each site in the network, the DTS configuration file defines all queues to be managed at a particular site. Separate threads are spawned to manage the queues individually and to allow queues to operate in parallel. Semaphores are used to indicate whether the DTS site and/or individual queues are allowed to proceed with transporting data. An RPC command may be issued to a DTSD instructing it to halt all processing, or just halt the processing of a particular queue. A queue will check both a primary DTS semaphore and an individual queue semaphore before processing any object for transport; each semaphore must be in an *active* state or the queue manager will block until both semaphores indicate processing can continue. RPC requests to halt a queue (or the DTS) therefore do not have an immediate effect in order to allow for an orderly shutdown of the system, however an *abort* method is provided that will immediately terminate processing.

Physically, a queue exists as a subdirectory of the *spool* area under the DTS root directory, one directory is created for each queue being managed. A file-based strategy is used to maintain the state of a queue and to provide recoverability in the event of an application or system crash. The top-level of the queue directory contains three files: the *status* file has a simple text string indicating the current state of the queue (‘active’, ‘processing’, ‘ready’, etc), the *current* file holds the identifier of the object currently being processed, and the *next* file contains the identifier to be used for the next data object submitted to the queue. Each of these files is updated as needed whenever a transport completes or new data are submitted to the queue; whenever a queue is activated a check for a failed (or *in-progress*) processing state is made and then recovered before normal processing begins.

Within the queue directory, an individual object being transferred exists in a subdirectory who’s name is a unique identifier assigned by the queue manager. Within that directory is the transfer-object file itself as well as another *status*

file used to maintain information about the state of processing of that object. A *control* file containing detailed information about the transfer object and its transport history is also created. This control file is initialized when the object is first submitted for transport and contains a list of keyword/value pairs giving the name of the submitting host, timestamps of each processing stage, file information (checksums, size, original name and directory path), arbitrary parameters added by the DTSQ invocation and a history record added by each DTS site in the path. During the transfer of an object, the contents of the control file are passed in the RPC method call (through an XML-RPC *struct* argument) rather than as a separate physical file, the destination site simply writes out a new copy of the control file when preparing the destination spool directory to receive the file. Once data have reached the end point of a data path, these control files may be collected by operations personnel to form a complete history of what was processed in the queue as well as characteristics about individual transfers; this history can be used to track the network performance of each leg of a data path to further tune the system.

5.1 Queue Types

Three *types* of queues are supported in the DTS: an **ingest** queue is typically the starting point for any data path and the target for the DTSQ application to submit data. It is special in the sense that *only* an ingest queue is allowed to modify the data: once data have been accepted into the spool area and the control file has been created, an ingest queue will execute the delivery application specified in the configuration file. This application is free to modify the file if desired (for our purposes we rely on this method to update FITS files with header keywords and checksum values needed for later archiving of the file, other file types are simply ignored for process will continue to be transported) and if it runs successfully, the control file is recreated since the file size and/or checksums may have changed. It is these recomputed values that are used to validate the integrity of the file during the remaining transfers along the queue path.

A **transfer** queue is the standard type for a queue in the middle of a data path; it is so named because it simply transfers data between machines. A transfer queue can be configured to deliver data as well, however the delivery application is run on the file copied to the delivery directory and not the version in the spool directory. An **endpoint** queue is defined for the last node in a data path; once the data have been delivered they are no longer needed in the spool directory and may be removed entirely from the queue.

5.2 Queue Modes

Queues operate in one of three defined *mode* types: A **normal** queue operates in a standard FIFO manner, becoming active and processing data anytime something new enters the queue and it remains active for as long there is unprocessed data in the queue. A **scheduled** queue is one that is active only at fixed intervals or at a specific time of the day. The scheduled activity is defined in the DTS configuration file either as an alarm (e.g. '*60m*' for an hourly cycle) or as a time of day at which to run (e.g. "*17:00:00*" for 5pm) and the queue will remain active until all data have been processed; intervals are calculated from the start of processing so as not to drift due to the processing time.

A **priority** queue is a special-purpose feature designed to halt processing of all other active queues so the full capacity of the network can be used to transport data in the queue, the block on other queues is released once processing is complete. This type of queue will not normally be used during routine observing and provides a simple means to override normal operations so that time-critical objects (e.g. GRB afterglow images, database updates that recover a crashed system to allow observing to resume, and so on) can be moved as quickly as possible ahead of other data waiting for transport. We expect an observer/operator to manually submit data to this type of queue when it is needed.

6. BOUNDARY INTERFACE APPLICATIONS

The DTS is expected to operate in a variety of settings: at the observatory where NOAO manages the observing environment and infrastructure, at the archive sites where the NOAO Science Data Management (SDM) personnel have operational control, and at collaborator's institutions (e.g. the DES pipeline at NCSA) where root system access is much more limited and we must interface to a completely unknown external system. In these diverse environments, it is not wise to integrate detailed knowledge of any one site's requirements into the overall DTS architecture.

A boundary interface to these external systems exists wherever data are *ingested* by the DTS for transport (e.g. at the observatory as part of data-taking, or at an archive site when replicating data between databases), and wherever they are *delivered* by the DTS to a machine in the network. Because the data processing requirements will often be unique to a

site, DTS uses a *delivery application* to do whatever processing of the file is required to interface to the site (an *ingest* of data by DTS is considered a *delivery* to the DTS, allowing us to recycle the concept depending on the context).

At an *ingest* site, this application is allowed to modify the contents of the file, at a *delivery* site the application is run on a copy of the file placed in the delivery directory specified by the DTS configuration file. This difference in behavior allows us to capture information about the site submitting the data without propagating a detailed environment specification. For archiving purposes, we rely on the ingest application to add the keywords necessary to uniquely identify the file in the database and establish the first archival copy of the raw data that will be moved through the system. On the delivery end, the application is free to do whatever is required and will not interfere with DTS operations. DTS delivery of a file acts as an event on the receiving machine, a typical response would be to notify an operator, take some action to stage the data for further processing, or to perform multiple or varied actions depending on the contents of the file being delivered.

6.1 Parameter Mechanism

The DTS will execute the delivery application in a child fork of the queue management thread. The configuration file defines this command using parameter tokens that are substituted for values from the object's control file before execution (e.g. a *'$Q'* for the queue name or a *'$F'* for the pathname to the delivered file). If no delivery application is specified then data are simply copied to the *delivery directory* without processing. This puts the onus of interfacing to external entities on the programmers responsible for accepting the DTS delivery and requires them to supply the application that satisfies the requirements of the DTS delivery for that site.

Table 1 lists the macros available for use in the DTS *deliveryApp* specification. An example delivery command definition would look something like:

```
/path/dtsDeliver –f $F –md5 $md5 ––rename $OP/$ON –inst $instrument
```

This executes the command *dtsDeliver* with the arguments needed to determine the file that has been delivered, its MD5 value so it can independently validate the file, and the original directory and filename so it can recreate the original directory structure if desired. In this example we show use of the generic parameter mechanism allowed in the control file. The DTSQ application allows arbitrary keyword/value pairs to be added to the control file when invoked, for example

```
dtsq –q mosaic –p instrument=mosaic /home/data/image001.fits
```

In this example, the `–p` flag defines an *instrument* parameter with a value of '*mosaic*'. When the delivery application is executed this value is available using the *$param* mechanism of the system. This allows arbitrary information to be made available when submitting data that might be useful to the receiving site without requiring special handling by the DTS itself.

Table 1. A listing of macro values allowed in the specification of the DTS delivery application. Before the command is executed, the values given

$F	Path to delivered filename	*$sum32*	32-bit SYSV checksum value
$S	Size of the file (in bytes)	*$crc32*	32-bit CRC checksum value
$E	UT epoch at time of submission	*$md5*	MD5 checksum value
$OF	Original file name		
$OP	Original directory path name	*$<param>*	Replaced by the value of named param
$OH	Originating host name		

Delivery applications may be compiled tasks or scripts; we require that the configuration file give the full path to the command to avoid assumptions about the execution environment. A *delivery policy* may be specified in the configuration as either *overwrite* or *renumber* to control the behavior of the DTS in the case of duplicate file names in the delivery directory. In *overwrite* mode the DTS will replace an existing file, in *renumber* mode an integer value is appended to the filename to make it unique.

7. PERFORMANCE

The parallel transport method implemented in DTS has significantly improved the network throughput we can realize over the network connections that in the past have been the most problematic. The natural variability of network conditions and our desire to not impact an operational environment meant we could not completely exercise the system in its final state, but spot-testing individual connections and using an average result has been encouraging.

Figure 4 shows the results of DTS transfers compared to other methods and the relative gains that can be achieved. Because we were using operational machines, it was not always possible to install the software needed to completely test other transport methods such as GridFTP, FDT, or a variety of UDP transport protocols in common use in the high-energy physics community.

The chart highlights the effect that network latency can have on the transport protocol chosen, and thus the need to be able to configure the system for the various network environments we'll encounter. The blue bars represent the movement of data from CTIO to Tucson over a 45 Mbps connection having a network RTT of ~300ms, the orange bars represent the KPNO to Tucson connection over a similar 45Mbps bandwidth link but with a RTT of only ~3ms, and the yellow bars represent a connection between two machines connected by a local gigabit ethernet. Since rsync is a very common general-purpose means of moving data, we've normalized the results to the performance of rsync over those connections. DTS testing was done using different numbers of processing threads, however it should be noted that these results can be affected by the speed and number of CPUs/cores available and in theory a more capable machine should be able to effectively handle more threads (either on a dedicated transport queue, or multiple queues being maintained simultaneously).

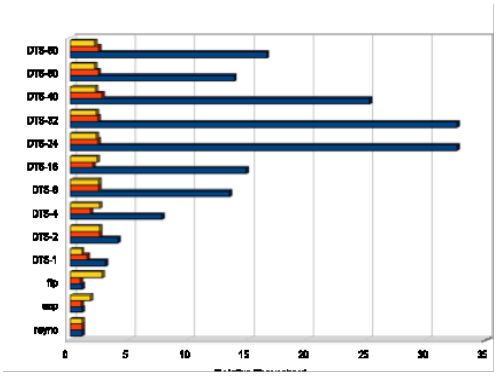

Figure 4. A graph of relative DTS performance

The results show that we can gain as much as a 32X increase in speed using DTS operating on ~24-30 sockets, but note we only see such a gain in performance over hi-latency connections. The performance increase is much less dramatic over lo-latency connections of similar bandwidth, however we don't require as many processing threads to achieve peak performance. The FDT streaming protocol in use by CERN for the LHC project was the only other method we were able to test directly and confirmed that DTS was able to achieve comparable throughput performance.

8. FUTURE WORK AND AVAILABILITY

Future work on DTS will be largely guided by experience gained operating the system in an actual observing environment. We expect to develop several new utility applications needed by operations staff and to add new features and capabilities as needed.

The DTS is in final test and is in the process of being deployed for use with the PreCam Survey being conducted at CTIO as part of the Dark Energy Survey science program. Please contact the author if you are interested in using the DTS at your institution.

REFERENCES

[1] http://www.xmlrpc.com/

[2] Seaman, R., Barg, I., and Zarate, N., "The NOAO Data Cache Initiative – Building a Distributed Online Datastore", Astronomical Data Analysis Software and Systems XIV, ASP Conference Series, Vol 347, 679-683 (2005).

[3] http://xmlrpc-c.sourceforge.net/

[4] http://curl.haxx.se/libcurl/
[5] http://www.doxygen.org/
[6] http://globus.org/toolkit/docs/3.2/gridftp/
[7] http://en.wikipedia.org/wiki/User_Datagram_Protocol
[8] http://monalisa.cern.ch/FDT/