Fast Data Transfer, a Microsoft Garage project

v1.0.0.7, Nov 2018

Introduction

Fast Data Transfer is a tool for transferring large amounts of data between your premises and Azure, by using the available network bandwidth as efficiently as possible.

Fast Data Transfer is a Microsoft Garage project. The Microsoft Garage is an outlet for experimental projects for you to try. Learn more at https://garage.microsoft.com.

The End User Licence Agreement (EULA) for Fast Data Transfer is here. It's also included in the client-side zip file and is installed along side the client executable.

Document road map

This document outlines a series of steps to get you up and running with Fast Data Transfer. The steps begin with a simple test mode. They progress to realistic client and server file operations, and end with moving to a signed SSL/TLS certificate.

This document also includes Tips for advanced usage, Frequently Asked Questions, instructions for upgrading, and advice on Performance Tuning.

For release notes, including change logs, see https://aka.ms/fast-data-transfer-update-info.

Support and suggestions

For support please email FastDataTransferHelp@microsoft.com. You can also visit http://aka.ms/fast-data-transfer-feedback to make suggestions for improvements.

When to use Fast Data Transfer

Fast Data Transfer has two components: the client-side software which runs on your premises, and a server-side component which runs in Azure VM(s) in your own Azure subscription. In contrast, most data upload tools load data directly from your client machine to Blob Storage. AzCopy is the best-known example. In many cases with small to moderate data volumes, direct-loading tools such as AzCopy are fine. You should step up to Fast Data Transfer if any of the following apply:

- You know, or suspect, that AzCopy is not fully utilizing the capacity of your network connection
- You have a very high-bandwidth link (e.g. 10 Gbps)
- You want to load directly to the disk of a destination VM (or to a clustered file system) without staging the data through Storage first. Direct Storage-loading tools, such as AzCopy, can't send data direct to VMs. Tools such as Robocopy can, but they're not designed for long-distance links and we have reports of Fast Data Transfer being up to 14x faster.
- You are reading from spinning hard disks, and want to minimize the overhead of seek times. (In our testing, we were able to double disk read performance by following the spinning disk tuning tips in Fast Data Transfer's instructions)
- You want to better understand the constraints on your upload performance – only Fast Data Transfer can show you whether the bottleneck is local disk, network, or destination storage. And only Fast Data Transfer allows you to address performance issues by changing the degree of parallelism separately at each stage of the process: reading from source, network transfer, and writing at destination.
- You want to throttle your transfers to use only a set amount of network bandwidth
- You need Fast Data Transfer's fully automatic retry-after-failure, rather than AzCopy's journaling-based approach.
- You need to transfer a large number of very small files (most direct-load tools don't handle this well)
- You have an ExpressRoute with private peering.

If you need to work with the *Files* and *Tables* features of Azure Storage, you should use AzCopy for those functions, as Fast Data Transfer only works with *Blobs*.

Installation Steps

The following steps guide you through the installation of Fast Data Transfer ("FDT" for brevity).

Install Server(s)

To use FDT, you need to install its server-side component on VMs in your target region - that is, the region where your data will reside. We supply an Azure Resource Manager template to create VM(s) and to install FDT on them. This section of the document describes how to use that template. (If, instead, you want to install the FDT server software onto existing VMs that you're already running, see Appendix: Installation on existing server VMs.)

1. Use the template to create one server VM for each 2.5 Gbps of expected throughput. E.g. Create 4 VMs if you want to get 10 Gbps of throughput. Create just one VM if your network bandwidth is 2.5 Gbps or less.

   To use the template, run it in Azure Portal by clicking this link: https://aka.ms/fast-data-transfer-run-arm-template. (If you prefer invoking templates with PowerShell, you can download AND this example of how to invoke it.)

   When you click the Portal link, you will be taken to a screen that looks like this (see image below). Note:

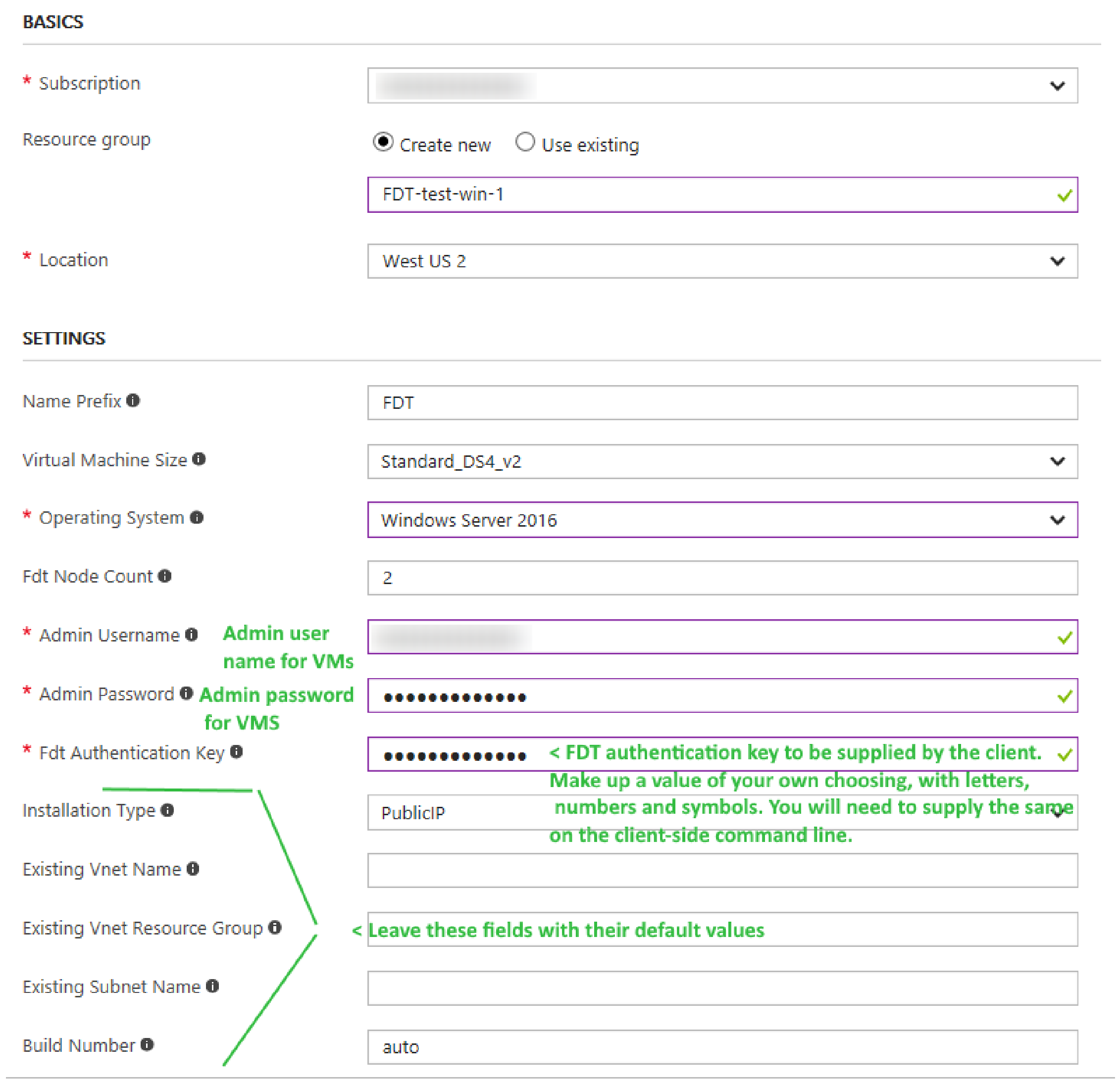
   – We recommend you choose **"Create new" Resource Group** in this release of the template.

   – Hover over the "i" symbols for help on the individual settings. This is particularly important for the **Virtual Machine Size**, where the help text will help you choose a suitable size.

   – A selection of operating systems is available in the drop-down list. If you wish to install on something that is not listed here -- different Linux distro or version, or a different version of Windows -- you will need to create the VM(s) manually and then go to Appendix: Installation on existing server VMs.

   – Fill out the form down to, and including, the "FDT Authentication Key" field. The **FDT Authentication Key** field is a value to be supplied by the client, to authenticate it with the server. Make up a value of your own choosing, including letters, numbers and symbols. It must have at least 16 characters in total. Remember the value that you choose.

   – NOTE: that there are two different password fields, one for the OS admin password and one for the FDT Authentication Key. They are NOT the same field repeated!



BASICS

* Subscription	[redacted]
Resource group	(● Create new ○ Use existing) FDT-test-win-1 ✓
* Location	West US 2

SETTINGS

Name Prefix	FDT
Virtual Machine Size	Standard_DS4_v2
* Operating System	Windows Server 2016
Fdt Node Count	2
* Admin Username	[redacted] ✓
* Admin Password	••••••••••••• ✓
* Fdt Authentication Key	••••••••••••• ✓
Installation Type	PublicIP
Existing Vnet Name	
Existing Vnet Resource Group	
Existing Subnet Name	
Build Number	auto

Admin user name for VMs

Admin password for VMS

< FDT authentication key to be supplied by the client. Make up a value of your own choosing, with letters, numbers and symbols. You will need to supply the same on the client-side command line.

< Leave these fields with their default values

2. At this point, the easiest option is to have the template automatically set up Azure networking for your VM(s), complete with a public IP address for each VM. To select this option, simply leave the Installation Type field at its default value, which is "PublicIP", and then skip all the remaining fields.

   – **"PublicIP"** tells the template to create a new Azure virtual network (VNET) for your VM(s), give each VM a public IP address, and set up a basic Network Security Group (NSG). The NSG allows traffic from all source IP addresses to reach your VM on the Fast Data Transfer port and on the admin port (RDP for Windows; SSH for Linux). After installation, you can manually tighten the NSG rules in the Azure Portal.

   – If you don't want the default behaviour, choose **"ExpressRoute"** as the installation type. This option works for *anyone* with an *existing VNET*, not just ExpressRoute customers. The template will put the VMs into the existing VNET and will not create an NSG or any public IP addresses. (If desired you can manually add public IP addresses later though the Azure Portal.) If you choose this installation type, then you also need to enter details of your VNET and a subnet within it.

3. Scroll down to the checkbox at the bottom to accept the terms and conditions. Check that box, then choose "Purchase".

4. The template will take about 5 minutes to run. It will create the VM(s), generate a self-signed SSL/TLS certificate on each server, then install and start the FDT server.

5. When the template finishes, you need to get connection details for the VMs out of the template's output, as follows:

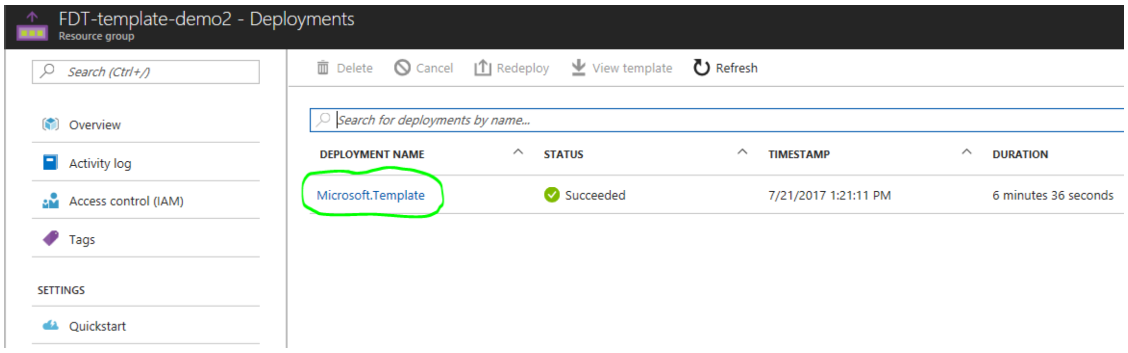
    1. In Azure Portal, open the Resource Group you deployed to.

    2. Choose the deployment named "Microsoft.Template".

    3. Then copy and paste the FDT connection strings from its "output" into a text file. There will be one connection string for each VM that was created.

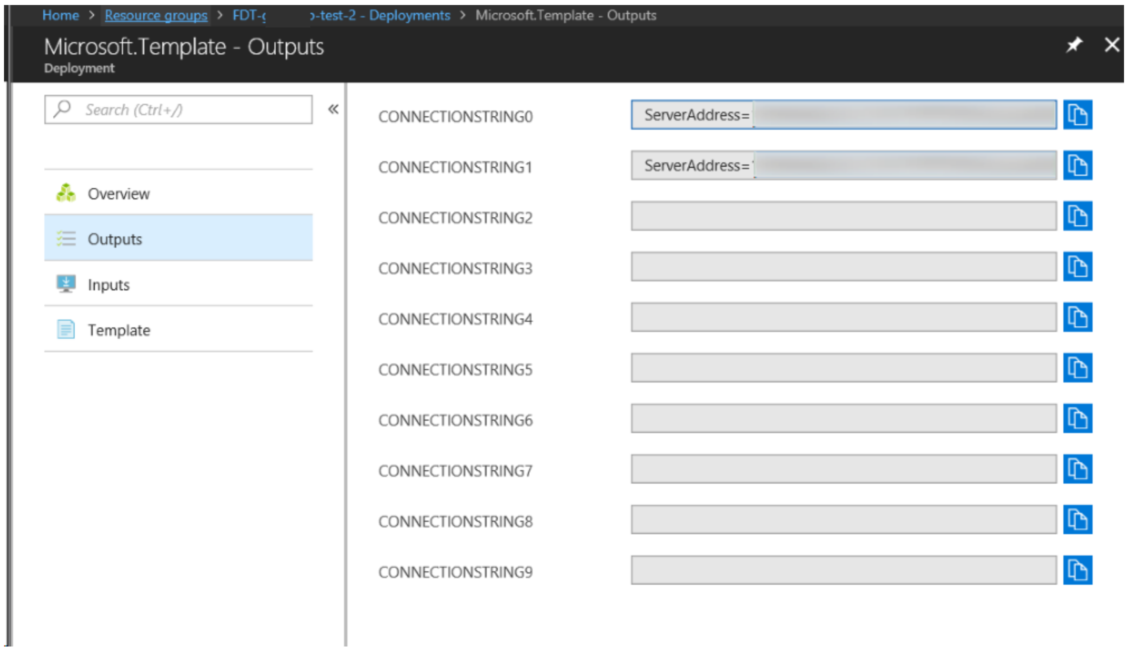
    4. Save the text file, so that you have a **permanent record** of the connection strings.

    Screenshot - Opening output of the deployment:



DEPLOYMENT NAME	STATUS	TIMESTAMP	DURATION
Microsoft.Template	Succeeded	7/21/2017 1:21:11 PM	6 minutes 36 seconds

    Screenshot - Getting the connection strings:


Home > Resource groups > FDT-( )-test-2 - Deployments > Microsoft.Template - Outputs

Microsoft.Template - Outputs

Deployment

CONNECTIONSTRING0	ServerAddress=
CONNECTIONSTRING1	ServerAddress=
CONNECTIONSTRING2	
CONNECTIONSTRING3	
CONNECTIONSTRING4	
CONNECTIONSTRING5	
CONNECTIONSTRING6	
CONNECTIONSTRING7	
CONNECTIONSTRING8	
CONNECTIONSTRING9	

6. Now that the template has successfully completed, the FDT Server will be running on each VM. On Windows, it runs as a service named FDT under user account Local Service. On Linux, it runs as a background process, viewable with `ps -ef | grep fdt` and running under an account named "fdt". See Tips for details.

7. If you used the default "PublicIP" installation type, this is a good time to go into the Azure Portal, open the Networking tab for your VM, find the RDP or SSH rule, and set its Source to be your own (on-premise) public IP address.

IMPORTANT On Linux, this release of the FDT Server does NOT automatically restart when the machine restarts. After a restart, you can run `sudo /opt/fdt-server/RunInBackground.sh` to restart the FDT server. On Windows, the FDT Service automatically restarts.

Install Client

To install the client:

1. Decide which machine you will use as your test client. You can:

   – **Use an on-premises machine.** This will show what FDT can accomplish using your own network link. For links rated at 1 Gbps or less, we recommend you use a machine with at least 4 CPU cores. For higher speeds, we recommend 8 or 16 cores.

   – **Use an Azure Virtual Machine in the cloud.** This is useful for testing FDT, since it will show you the upper bounds of what FDT is capable of on very fast networks. (But it's of no help in transferring the data that's on your premises!) Use an Azure VM with at least 16 cores, and locate it in a different Azure region from where you put the servers. (Putting it in the same region works, but makes it unrealistically "easy" for FDT to achieve fast performance. Region-to-region tests are more realistic.)

– **Try both.** This lets you can compare the performance you see over your own network, to that seen from a cloud VM. The performance you see from cloud VM is representative of what you should expect to see you if installed a 10 Gbps link to your own premises.

2. Ensure the necessary pre-requisites are installed:

   – Windows Server 2016: You do not need to do anything, because the required framework version (.NET Framework 4.6.2) or later is already installed.

   – Windows Server 2012 R2: install .NET Framework 4.6.2 or later. (If you're not sure whether its already installed, see instructions here or simply try to install it. The installer will tell you if it, or a later version, is already installed.)

   – Linux: FDT runs under .NET Core 2.0, which supports most major Linux distros. You do not need to install .NET Core, because the necessary files are included with the FDT application. You do, however, have to install the native dependencies of .NET Core, which are the standard Linux packages that .NET Core depends on. Options for installing them are listed in Appendix: Installation of Linux dependencies, but the easiest is to use your distro's standard package manager, as follows:

     • Fedora and CentOS: `sudo yum install libunwind libicu`

     • SUSE Linux and OpenSUSE: `sudo zypper install libunwind libicu`

     • Ubuntu and Debian: `sudo apt-get install libcurl3 libunwind8`

3. Download the FDT client to your machine:

   – Windows: In PowerShell
   ```
   wget "https://aka.ms/fast-data-transfer-windows-client" -Out FDT-client.zip
   ```
   or else manually download the zip file.

   – Linux:
   ```
   wget "https://aka.ms/fast-data-transfer-linux-client" -O FDT-client.tar.gz
   ```

4. Unzip the client into a directory of your choosing:

   – Windows: right click it and choose Extract All.

   – Linux:
   ```
   mkdir fdt-client
   tar -xf FDT-client.tar.gz -C fdt-client
   ```

5. Update the client config file. To do this:

    1. Use any text editor to open the file FastDataTransferClient.settings.json, which you will find in the FDT directory.

    2. Find the line for ConnectionStrings, and remove its leading slashes to uncomment it. (See screenshot below.)

    3. Put in the connection strings that you got from the server-side deployment. You will have one connection string for each FDT server. (Again, see screenshot below.) Each connection string must be surrounded by double quotes, and they must be separated by commas. For readability, you can add line breaks between the strings.

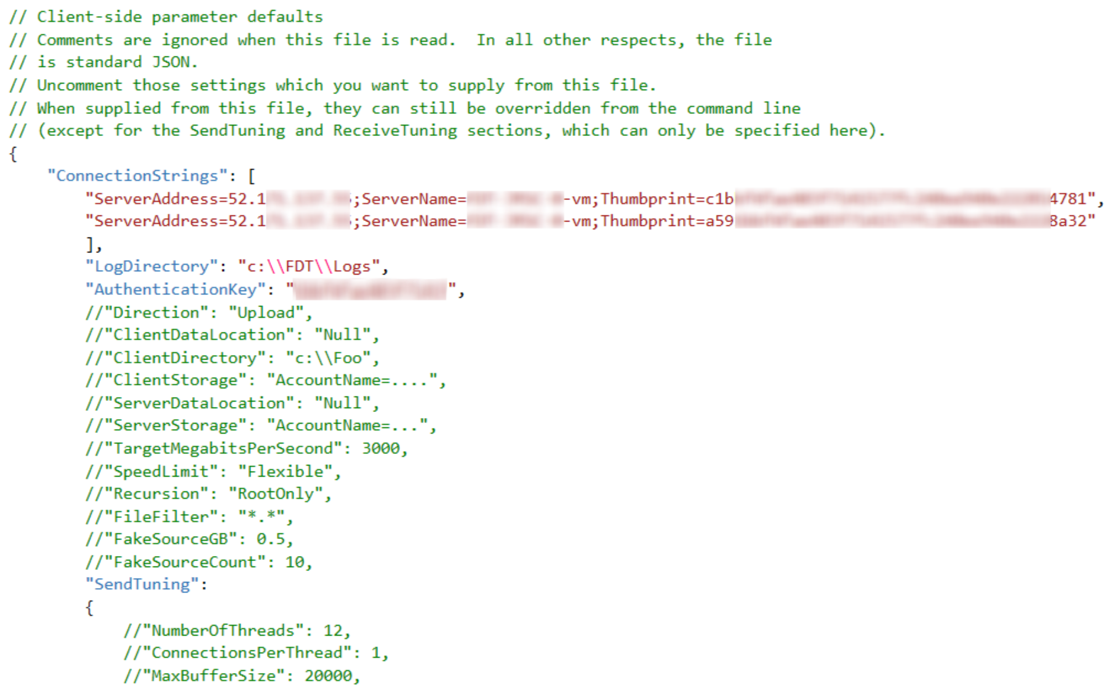
    4. Uncomment the AuthenticationKey line, and set its value to be the Authentication Key that you chose when installing the server. **NOTE** if you are using Windows, then for added security you can encrypt the key with DPAPI. See How can I encrypt the authentication key?.

    5. Also uncomment the LogDirectory line, and fill in the location of a directory where the FDT client can save its log files. On **Windows** you MUST use **double** backslashes in the path (as shown below).

    6. Save the file.

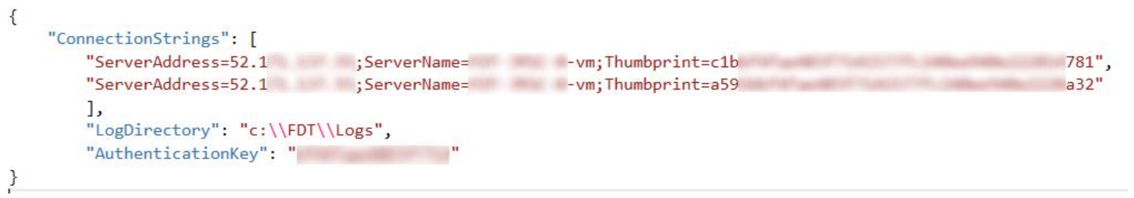
    7. Set OS-level permissions on the file, so that it can only be read by the user(s) who will run FDT. (Securing the file is important, because it contains the FDT AuthenticationKey).

   Your completed file should look something like this:



```
// Client-side parameter defaults
// Comments are ignored when this file is read.  In all other respects, the file
// is standard JSON.
// Uncomment those settings which you want to supply from this file.
// When supplied from this file, they can still be overridden from the command line
// (except for the SendTuning and ReceiveTuning sections, which can only be specified here).
{
    "ConnectionStrings": [
        "ServerAddress=52.1[redacted];ServerName=[redacted]-vm;Thumbprint=c1b[redacted]4781",
        "ServerAddress=52.1[redacted];ServerName=[redacted]-vm;Thumbprint=a59[redacted]8a32"
        ],
        "LogDirectory": "c:\\FDT\\Logs",
        "AuthenticationKey": "[redacted]",
        //"Direction": "Upload",
        //"ClientDataLocation": "Null",
        //"ClientDirectory": "c:\\Foo",
        //"ClientStorage": "AccountName=....",
        //"ServerDataLocation": "Null",
        //"ServerStorage": "AccountName=...",
        //"TargetMegabitsPerSecond": 3000,
        //"SpeedLimit": "Flexible",
        //"Recursion": "RootOnly",
        //"FileFilter": "*.*",
        //"FakeSourceGB": 0.5,
        //"FakeSourceCount": 10,
        "SendTuning":
        {
            //"NumberOfThreads": 12,
            //"ConnectionsPerThread": 1,
            //"MaxBufferSize": 20000,
```

If you prefer a simpler file, you can remove all commented lines (those starting with //) to strip the file right back to something just a few lines long:



```
{
    "ConnectionStrings": [
        "ServerAddress=52.1[redacted];ServerName=[redacted]-vm;Thumbprint=c1b[redacted]781",
        "ServerAddress=52.1[redacted];ServerName=[redacted]-vm;Thumbprint=a59[redacted]a32"
        ],
        "LogDirectory": "c:\\FDT\\Logs",
        "AuthenticationKey": "[redacted]"
}
```

Prove Connectivity

To check that your client can connect to the server, do a small data transfer.

1. Run the following command line. It will transfer 1 GB of randomly-generated data:

   Windows:

   ```
   FastDataTransferClient.exe -n Upload -c Null -s Null --FakeSourceGB 1 -M 1000
   ```

   Linux:

   ```
   ./FastDataTransferClient -n Upload -c Null -s Null --FakeSourceGB 1 -M 1000
   ```

   The purpose of each command line parameter is as follows:

   – -n Upload says that this transfer will be an upload.

   – -c Null tells the **c**lient to that it should not connect to any real data store. In FDT "Null" means "If you're sending data, just generate fake data. If you're receiving, just throw it away". Since the client is *sending* when uploading, it will generate fake data for this test.

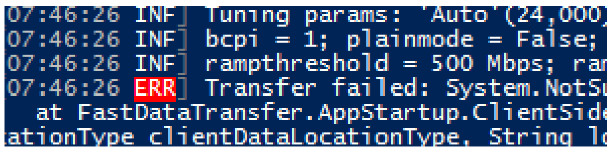
- -s Null tells the **s**erver that it should not connect to any real data store. Since the server is *receiving* in an upload, it will just throw the received data away.
- --FakeSourceGB 1 says that when generating fake data, the quantity generated should be 1 GB.
- -M 1000 stands for 1000 Mbps and indicates that FDT should aim for a speed of 1000 Mbps in this transfer. Below, in the section Set Target Speed, we will refine the value of this parameter. But for now, 1000 will be fine.

A full command line reference table can be found below.

2. When the command runs, you'll see activity on the client as it displays messages. Messages will also be written to the Log directories on client and server.

3. Wait for the transfer to complete, then check that it succeeded. It succeeded if the final line of the output says something like this:
"All 200 files in this transfer have been successfully received and saved. Their checksums matched between sender and receiver."
In the unlikely event that the transfer fails, you will see an error message instead. The error message will start with a red "ERR" indicator, as shown in the screenshot below. The first line of the error message will indicate the cause of the problem. In most cases, that will give you enough information to fix the problem and successfully re-run the transfer. (If the problem can't be resolved, please contact us for assistance.)

   Screenshot of on-screen output, showing error indicator:



```
07:46:26 INF] Tuning params: 'Auto'(24,000)
07:46:26 INF] bcpi = 1; plainmode = False;
07:46:26 INF] rampthreshold = 500 Mbps; ra
07:46:26 ERR] Transfer failed: System.NotSu
  at FastDataTransfer.AppStartup.ClientSide
ationType clientDataLocationType, String lo
```

Set Target Speed

To perform optimally FDT needs an initial target speed that is realistic for your network. This initial target speed is measured in mega *bits* per second, and is supplied as the value of the -M parameter. Here's how to find the right value for your environment:

1. When tuning the target speed, be sure that the -c and -s parameters are both set to Null. That means FDT will transfer randomly-generated without touching the file system, and therefore the results will reflect the true capabilities of the network. But **only use 'Null' for testing. Never use 'Null' when trying to move real data.**

2. If you know your network is faster than 1 Gbps, then compute: 2750 x (number of FDT servers). For example, with 4 servers this gives you 11000. Run a test using that as the

value of the -M parameter on your client, and FakeSourceGB of 50 (or higher). E.g.
`FastDataTransferClient.exe -n Upload -c Null -s Null --FakeSourceGB 50 -M 11000`
If you're happy with the speed this achieves, then you're done. Remember the value of -M, so you can use it from now on, then skip ahead to Send Real Files.

3. Otherwise, run a test *without* the -M parameter. Omitting -M puts FDT into a mode where it tries to agressively seek the right speed by all by itself. E.g.
`FastDataTransferClient.exe -n Upload -c Null -s Null --FakeSourceGB 10`

4. If it takes less than 2 minutes to complete the transfer, double the FakeSourceGB parameter and repeat the test. Continue doubling and repeating until you have a test that takes at least 2 minutes.

5. Now multiply the FakeSourceGB parameter by 5, so your next test run will last for about 10 minutes.

6. Run the tool and wait until the "10 minute" transfer finishes.

7. In the client side output, the second-to-last line will begin with the words *"Target rate recommendation"*. It will say one of the following:

   – *"You might get better performance if you supply a target value of ... Mbits/sec"*. Take the suggested value as the value of your -M parameter.

   – *"There is not enough information to recommend which Mbits/sec target you should specify on the command line"*. This message should be very rare, after a 10-minute run. If you get it, we'd appreciate you sending us a copy of the client-side log file. Then, we suggest you look at the final *target* speed shown in the log (in the tabular section of log output just above the final messages). Take that value as the value of your -M parameter.

Remember, **make sure you do include the -M parameter in all your normal runs of the tool**. Only leave it out when seeking an optimal rate. Note also that -M specifies a *target*. FDT's actual throughput is always slightly below the target.

Send Real Files

To send real files, from the client-side file system, edit the client-side command line:

- Remove `--FakeSourceGB nn`
- Change `-c Null` to `-c Disk` to tell the client to read from disk
- And add `-d <sourceDirectory>` to say where to read the data from

Windows example: `FastDataTransferClient.exe -n Upload -c Disk -s Null -d c:\foo\bar -M 11000`
Linux example: `./FastDataTransferClient -n Upload -c Disk -s Null -d /foo/bar -M 11000`

By default, FDT sends all files in the specified directory, but does not process subdirectories. If you need to recurse into subdirectories, add `-r Recurse` to the command line. To include only certain files, specify a filter wildcard with the -f parameter. E.g. `-f A*.dat` would include files A1.dat and Abc2.dat, but not A1.txt or B1.dat.

If the transfer speeds you see now are just as fast as those you saw when setting the target speed, then you know your client-side disks are fast enough. But if the speeds you see now are significantly lower, see the Tuning section below for advice on client-side disk tuning.

Once you have done a few test runs, with the client sending real files, you are ready to move on to Saving to Azure Storage.

Save to Azure Storage

If your client disks were fast enough (above) then leave -c and -d as they are. However, if your client disks were too slow, then temporarily change to `-c Null`, remove the -d parameter and reinstate `--FakeSourceGB 10`. That way the client will generate fake data, without using disk, allowing you to focus your attention on the server-side aspects of performance.

To save to an Azure Storage account, edit the client-side command line as follows:

- Change the -s parameter to `-s Storage`
- Add `--ServerStorage` followed by a connection string that describes how the server should connect to storage.

E.g.

Windows:

```
FastDataTransferClient.exe -n Upload -c Disk -s Storage -d c:\foo\bar -M 11000 --ServerStorage
"AccountName=myaccount;AccountKey=A1B3...;ContainerName=mycontainer"
```

Linux:

```
./FastDataTransferClient -n Upload -c Disk -s Storage -d /foo/bar -M 11000 --ServerStorage
'AccountName=myaccount;AccountKey=A1B3...;ContainerName=mycontainer'
```

Files from the client-side directory will be uploaded as blobs into the container specified in the Storage connection string. If recursion is enabled with `-r Recurse` then client side subdirectories will be created as virtual directories in Storage.

The connection string is similar to those provided for Storage accounts by the Azure portal, but it contains additional elements. The elements of the Storage connection string are:

Name	Required	Description	Comment
AccountName	Y	Name of Storage account	As per standard

			Azure Storage connection strings
AccountKey	Y	"Shared Key" of Storage account	As per standard Azure Storage connection strings
ContainerName	Y	Name of the container to upload to. Container will be created if it does not already exist	FDT-only
DefaultEndpointsProtocol	N	https (default) or http. Specifies the protocol to be used between FDT Server and Azure Storage	As per standard Azure Storage connection strings
EndpointSuffix	N	Defaults to core.windows.net	As per standard Azure Storage connection strings
BlobType	N	Block (default) or Page. Note that Azure Storage requires all page blobs to have sizes that are multiples of 512 bytes AND you should only use Page blobs for VHD files.	FDT-only
BlockSizeInKB	N	Defaults to 8192	FDT-only
StoreBlobHashes	N	Should the Content-MD5 field be populated in Storage for each blob? Defaults to false. The original beta release defaulted this to true, but it now defaults to false. Before relying on this parameter, please note that for technical and performance reasons, future versions of FDT may reduce support for using true.	FDT-only

In rare cases where the default endpoint settings are not suitable, and you don't want to use DefaultEndpointsProtocol and EndpointSuffix, you can instead specify the full blob endpoint URL as the value of 'Endpoint'.

How to encrypt Storage Credentials

As of version 1.0.0.5, FDT allows you to encrypt the Storage credentials, instead of providing them in plain-text on the command line. This provides added security for your Storage Account Key. See Can I encrypt the Storage Credentials in the FAQ section.

Linux performance when saving to Storage

Previous versions of FDT were significantly slower saving to Storage when the server was on Linux, compared to when the server is on Windows. As of version 1.0.0, the difference is greatly reduced. If you still have any problems with performance to Storage on Linux, the following may help:

- Experiment with reducing the degree of parallelism in the traffic to Storage, by adding this setting to the server-side config file (FastDataTransferServer.settings.json): `"MaxWritersPerCpu": 1`. Remember that all settings in the file, except the last, must be followed by a comma, and that you must restart the FDT server program after changing the settings. You can restart it with these two commands:
  ```
  sudo /opt/fdt-server/ShutdownInBackground.sh
  sudo /opt/fdt-server/RunInBackground.sh
  ```

- Run an extra server. E.g. to get the same performance to Storage, you might use 4 Linux servers but only 3 Windows ones.

- Run your FDT servers on Windows. Linux FDT clients can connect to Windows servers.

Finally, in the future we plan to upgrade FDT to .NET Core 2.1. At that time we expect additional improvement of Linux performance to Storage, due to the resolution of this HTPS performance issue.

What about saving to Disk on the server?

See Using server-side disk in the Tips section, below.

What about saving to Azure Files?

See saving to Azure files in the FAQ section, below.

Finalize Client-Side Parameters

Now you have tested all the major elements of FDT in your environment. We suggest you review your command line parameters, to make sure FDT is set up as you need it. Here's a summary of the available parameters.

Note For all parameters, you can use the full name on the command line, with a *double* dash in front of it. Some parameters also have a short form, which is preceeded by a *single* dash. E.g. --ClientDataLocation Disk or -c Disk.

Full Name	Short Form	Required	Description	Allowed Values
Direction	-n	Always	Direction of transfer	Upload, Download
ClientDataLocation	-c	Always	Type of location where client should read/write data. "Disk" is the most common value. See Prove Connectivity for explanation of "Null".	Null, Disk
ClientDirectory	-d	If -c is Disk	Client side file system directory. Client reads from here if uploading and writes to here if downloading.	Fully-qualified path to a directory
ServerDataLocation	-s	Always	Type of location where server should read/write data. "Storage" and "Disk" are the most common values. See Prove Connectivity for explanation of "Null".	Null, Disk, Storage
ServerStorage		If -s is Storage	Storage connection string as described above. Server writes to here if uploading to Storage, and reads from here if downloading from Storage.	
TargetMegabits PerSecond	-M	Only omit if probing for optimal rate	Target rate, at start of transfer. FDT will make refinements to this rate as the transfer proceeds.	
SpeedLimit	-t	Optional	Controls whether speeds higher than -M may be attempted by	Flexible (default), Fixed

			FDT's speed-tuning algorithm. Setting to Fixed prevents speeds above -M.	
Recursion	-r	Optional	Controls whether sender recurses into subdirectories (or virtual directories if downloading from Storage)	RootOnly (default), Recurse
FileFilter	-f	Defaults to *.*	Wildcard pattern to select which files to send. If downloading from Storage, FileFilter may include at most one * and if present that * must be at the end. See How do the filter wildcards work below.	
FakeSourceGB		May be used if -c is Null	Number of GB of test data to generate when: (Uploading with -c = Null) OR (Downloading with -s = Null). May include decimal point, e.g. 0.5.	Any number. Defaults to 10.
FakeSourceCount		May be used if -c is Null	Number of files into which FakeSourceGB should be broken	Any positive integer. Defaults to 200.
LogDirectory	-L	Optional	Generally specified in FastDataTransferClient's settings.json file instead of on command line. If not specified, *no* logging will be performed. Can be fully-qualified or may be a relative path starting with dot-slash. The latter will make it a sub-directory of the FDT client application	Path to a directory.

			directory.	
LogCompletedFiles		Optional	If set to true then additional log files are created in the LogDirectory. They have names starting with 'completed-files...' and they list the names of all files that have been successfully transferred.	Can be true or false. The default is false
FileDatesOption		Optional	If set to Preserve then each file's creation date and last modified date will be copied across to the destination, so that file dates on the destination are the same as those at the source. The copied dates and times are correctly adjusted for any time zone differences between sender and receiver. NOTE: At its introduction in version 1.0.0.6, this option is only supported for uploads, and only when uploading to Disk (not to Storage).	Can be Discard (default) or Preserve. If Discard, then file dates at the destination will be set to the date and time when the transfer took place.
AuthenticationKey	-a	Always	Generally specified in FastDataTransferClient's settings.json file instead of on command line.	
AuthenticationKey ProtectionType		Optional	Says how AuthenticationKey has been encrypted.	Can be None (default), User or Machine
DiskReadBuffer Kilobytes		Optional	Specifies the buffer size to use when reading files from disk, in kB. In some cases a larger size may improve disk read	The default value for this parameter is 4 kB.

		performance. Note that the actual buffer size used is the greater of this value and the 'FDT Buffer Size', which applies to network traffic and is specifed in the 'Buffer Sender' section of the config file. See Tuning.	
UnauthorizedFileAction	Optional	Says how FDT should respond if it tries to read a *directory* that it does not have access to. (As of release 1.0.0.5, this setting only affects directories. Individual files are always skipped and logged, if they cannot be read.)	Can be Error, Skip or SkipAndLog. Defaults to SkipAndLog
RunDateLogFile	Optional	Used to transfer only modified files. See below	
FileNames	Optional	Alternative to FileFilter. Is a space-delimited list of filenames to transfer. Must be fully-qualified names, and files must be somewhere under ClientDirectory. Do NOT use with RunDateLogFile.	
FileNamesFile	Optional	As for FileNames but contains the name of a file that lists the files to transfer, one per line. Do NOT use with RunDateLogFile.	
ExclusionFile	Optional	If specified, transfer will exclude all directories whose full names END WITH anything listed in the file. It is case sensitive. When a	

		directory is excluded, all its children are excluded too.	
CommandLine ParametersToLog	Optional	Space-delimited list of command line parameters that should be included in the log.	Can include FileSpec (directory and filter), StorageContainer (account name and container name), ServerConnectionStrings. In version 1.0.0.5, the latter only works if server connection strings were specified on the command line (see below).
TelemetryType	Optional	Controls whether to send telemetry (anonymous FDT performance statistics) to Microsoft. Generally specified in FastDataTransferClient's settings.json file instead of on command line.	Can be Send (default), None (to supress sending), or SendAndLog which both sends the telemetry and puts the same data into the client-side log file.
NotifyOfUpdates	Optional	Can be true (default) or false	Controls whether FDT should check to see if its up-to-date. If it checks, and is out of date, it will display a warning as described below

It is also possible to put the FDT Server Connection Strings on the command line. The parameter short name is -k and it should be followed by a space-delimited list of connection strings. However, it is generally easier to put these connection strings into the FastDataTransferClient.settings.json file instead, as described in these instructions.

Finally, if you want to *only* check to see if FDT is up to date, without running any transfer at all, supply --TestUpToDate on the command line with no other parameters. See below for details.

Use Real Certificate

So far, to secure the connection between the FDT client and the FDT server, you have been using self-signed SSL/TLS certificates. The certificates, one per FDT server, were generated by the Azure Resource Manager template when it created the servers.

The thumbprints of the certificates are included in the FDT Server Connection Strings, which you saved in FastDataTransferClient.settings.json. The FDT client will only accept self-signed certificates that have thumbprints matching those in its Connection Strings.

For production use, it is best to move to a real certificate because:

- The issuing Certificate Authority takes responsibility for keeping abreast of security standards (so each time you renew, the certificate's properties will reflect latest best-practice).
- You can leverage standard mechanisms for any certificate revocation that may become necessary.

To move to a real certificate:

1. Register a domain name, if you don't have one already.
2. Configure DNS settings so that each of your FDT servers has a hostname within your registered domain. E.g. if your domain is `contoso.com` your FDT servers might be `fdt1.fdt.contoso.com` and `fdt2.fdt.contoso.com`.
3. Obtain SSL/TLS certificate(s) from a Certificate Authority of your choice. You may choose to get a wildcard certificate for all your FDT servers e.g. `*.fdt.contoso.com`. We suggest that any wildcard certificate should apply only to a subdomain that includes your FDT servers, e.g. `*.fdt.contoso.com`, rather than your whole enterprise, `*.contoso.com`. (This suggestion is for defense-in-depth against any compromise of private keys.)
4. Record the thumbprint of the certificate (you will need this below).
5. Package the certificate, plus the corresponding private key, into a PKCS #12 file, with *no* password required for the private key. (Instead of a password, access to the private key will be controlled by access rights that you will set in the next step). On Windows, PKCS #12 files are usually saved with the extension .pfx, and you can pack a certificate and private key file into a .pfx file using Pvk2Pfx. On Linux you can use OpenSSL's `pkcs12` command.
6. Install the PKCS #12 file on each FDT server, and **set permissions to control access to its private key**:
   - Windows:

   1. "Manage Computer Certificates" (under Control Panel) then install the certificate into Personal -> Certificates folder. See screenshot below.

   2. Then you **MUST** right-click the certificate, choose All Tasks -> Manage Private Keys; and give Read access to the account that FDT runs under (which is the 'Local Service' account by default).

 – Linux: The file must be put in the location where .NET Core apps will look for it. Assuming FDT runs under its default account of 'fdt', this is as follows: cp <cert>.pfx /home/fdt/.dotnet/corefx/cryptography/x509stores/my
chown -R fdt: /home/fdt/.dotnet/corefx/cryptography/x509stores/my

7. Log into each server and edit the FDT server-side settings, to tell FDT to use the new certificate. To do this, edit the file FastDataTransferServer.settings.json in any text editor. Set the TlsCertificateThumbprint value to the thumbprint of your new certificate.

8. Edit your client-side settings.json file, altering each connection string as follows:

 – Remove Thumbprint=... from each connection string. (The client does not need it for real certificates).

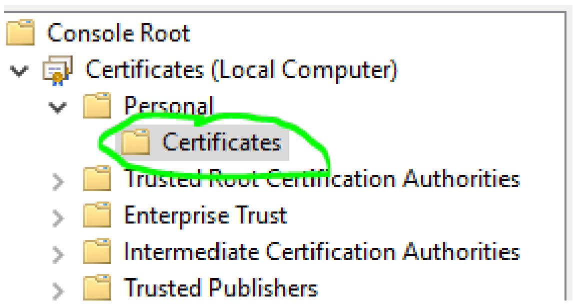
 – Remove ServerName=... from each connection string.

 – Change the value of ServerAddress to be the fully-qualified domain name of the server. E.g. fdt1.fdt.contoso.com

9. Run a small transfer to test connectivity.

10. **Important:** Put in place reminders and procedures to renew your certificates before they expire.

Screenshot - Certificate folder on Windows:


Console Root
Certificates (Local Computer)
Personal
Certificates
Trusted Root Certification Authorities
Enterprise Trust
Intermediate Certification Authorities
Trusted Publishers

Tips

Using server-side disk

When initially installed, the FDT server can connect to Azure Storage and to "Null" (i.e. throw data away, to test connectivity and network speed).

It can also access the VM's file system, for uploads and downloads, if you enable the capability as follows:

1. On each FDT server, go to the FDT installation directory. On Windows this is c:\FDT\Server and on Linux it is /opt/fdt-server.
2. In that directory, open the file FastDataTransferServer.settings.json, and
    1. Add "Disk" to the set of LocationTypes
    2. Add "DiskLocation" to be the directory when you want data to be uploaded to. On Windows you **MUST** use **double** backslashes in the path, because all backslash characters must be escaped (i.e. doubled) in JSON files. (Linux's forward slashes do not need to be doubled.)
    3. The relevant lines of your finished config file should look something like this:
    ```
    "LocationTypes": ["Null", "Storage", "Disk"],
    "DiskLocation": "D:\\yourDirectory",
    ```
    4. Save the file.
3. Review file system permissions (access control) for the FDT server process. On Linux, by default the process runs under a user named 'fdt'. On Windows, you can see which user is used by looking at the properties of "Fast Data Transfer" under Services in the Control Panel. On both operating systems, the user needs read and execute access on its own program directory, and read and write access to the `DiskLocation` that you set up in the config file. For defense in depth, it should not have broad read or write access to other parts of the file system.
4. Restart the FDT Server:
    - Windows: Run `RestartService.cmd` or restart the Fast Data Transfer service from the Windows Control Panel's "Services" app.
    - Linux: Run
    ```
    sudo /opt/fdt-server/ShutdownInBackground.sh
    sudo /opt/fdt-server/RunInBackground.sh
    ```
5. On the client-side command line, use `-s Disk` to indicate that the server should use disk for the transfer.
6. Now you can upload to, and download from, the server's file system.

For security reasons, the client app cannot override the directory you specify in the server's JSON file. I.e. when uploading, the client cannot force anything to be saved outside that directory; and when downloading, it cannot read anything outside that directory. If the client uses -r Recurse command-line option, then the sender-side subdirectories will be re-created inside the receiver's target directory.

NOTE: If you need to do a number of different transfers to different server-side directories, you have two options:

1. One is to transfer to different subdirectories of a common root. That works well if you already have the data as different subdirectories of a common root on the client side. (See next section of this document for details.)

2. Your other option is to manually edit the server's FastDataTransferServer.settings.json file after each transfer, to point it to a different location. Remember to restart the FDT service after each edit. In this scenario, you will normally only want one successful transfer to go into each target directory. I.e. the sequence is transfer data -> edit server config -> transfer more data -> edit server config, and so on. You can enforce that sequence, and make sure no-one accidentally runs *two* different transfers into the same target directory, by setting DiskLocationCompletionLockFileName in FastDataTransferServer.settings.json. When that setting exists, a file with the given name is created in the root target directory after a successful transfer. Future transfers, if configured to use the same DiskLocationCompletionLockFileName, will refuse to transfer data as long as this file exists in the root target directory.

   – For example:
     "DiskLocation": "D:\\Data",
     "DiskLocationCompletionLockFileName": "_DataTransferCompleted_Lock.txt",

Symbolic links: If the target directory contains any symbolic links (Windows and Linux) or junctions (Windows), pointing to directories outside it, they will appear to FDT to be directories *within* the target, and so will be accessible for uploads and downloads.

See Tuning below, if performance is slower than expected when saving to server-side disk.

Downloading

To download, instead of upload, use -n Download on the client-side command line. Any file filter (-f) or recursion (-r) settings will be applied on the *server* side when downloading.

Just as when uploading, the *client's* -s parameter specifies whether the server should access Null, Storage, or Disk. 'Null' means that the server should generate fake data, to be downloaded. 'Storage' means that the server should read from the location identified by the --ServerStorage parameter. 'Disk' means the server should read from the DiskLocation specified in its config file. 'Disk' only works if server disk access has been enabled, as described below.

For speeds above 1 Gbps, make sure that you adjust the following settings on your client machine:

- Windows: Maximize the size of the network adapter's receive buffer. The exact setting varies depending on the adapter: some display a total amount of RAM, while others display a count of "receive descriptors". In any case, you should choose the maximum available value. You should also check that Receive-Side Scaling (RSS) is enabled. The RSS and buffer size settings can be found by going into Control Panel -> Network & Sharing Center -> Change Adapter Settings; then right clicking the adapter and choosing Configure.

- Linux: Ensure that the OS settings are adjusted for relatively large amounts of receive buffering. If you're not familiar with the settings that may be relevant, you can copy some or all of the settings that we use on FDT Linux servers. You can find those settings in the FDT Server Linux setup script. (Don't run the whole script on the client! Just find and copy the relevant settings.) The script is:

  https://aka.ms/fast-data-transfer-setup-linux-server

IMPORTANT Only one server can be used in a download transfer. When downloading, the client will use the first server listed in its config file, and ignore the rest.

Transferring only changed files

Currently this feature is only supported when transferring from client-side disk to Azure Blob Storage or to Server Disk.

WARNING If you use this feature when transferring to Server Disk, is it up to you to make sure that the server's DiskLocation has not been changed between runs. In other words, when deciding which files to transfer, the client will look at the RunDateLogFile, and say "I last transferred files to this server on date X, therefore this time I'll only transfer those that have changed since date X". If, in the meantime, someone has pointed the Server's DiskLocation to a different drive, then the client's behaviour is probably not what you want... but the client has no way of knowing that. So use with care when saving to Server Disk.

To transfer only changed files:

Add the client side parameter `--RunDateLogFile ...` and specify, as its value, the path to a file. E.g. `--RunDateLogFile c:\FDT\RunDates.txt`. The file should not initially exist because FDT will automatically create it.

After each sucessful run, FDT will record into that file the following information: the source directory; the destination Storage account name and container name(*); and the time when the FDT transfer started. When you run FDT again with the same RunDateLogFile, and the same source directory, destination account and container, it will only transfer files which have been modified *after* the previous successful run. The date in the file is updated after each successful run so that you can run FDT repeatedly, and each time it will pick up only the files that have changed since the previous run.

(* When transferring to Disk, instead of Storage, it will record the Server's name and IP address)

Note that, to accomodate small differences in clocks, there is one minute of leeway built in. Specifically, FDT will pick up files which were modified after: (StartTimeOfPreviousRun - 1 minute).

Transferring part of a directory tree

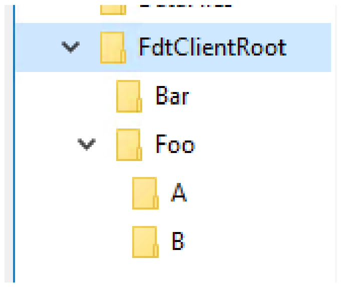
Sometimes, your source directory may contain many child files, and you only want to transfer some of them. (E.g. for a smaller-scale test, or to break up a very big transfer into smaller jobs). Here's an example to show you you can do that:

Server config:

(Note the **double** backslash; any Windows path included in a JSON file must be modified in this way because all backslash characters must be escaped (i.e. doubled) in JSON files.)

```
"LocationTypes": ["Null","Storage", "Disk"],
"DiskLocation":"D:\\FdtServerRoot",
```

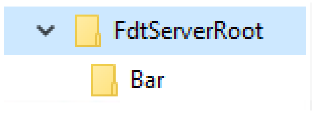
Client side file structure:



Note that there are subdirectories of the client root directory. We can transfer just ONE of those subdirectories, as follows:

```
.\FastDataTransferClient.exe -n Upload -c Disk -s Disk -d d:\FdtClientRoot -r Recurse -f Bar\*
```

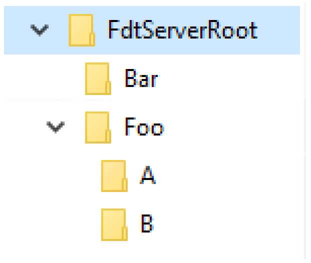
Note that we nominate the FdtClientRoot directory in the -d parameter, then we tell FDT to recurse to subfolders, and then we set the filter (-f) to filter it to only the things in one child directory. After running this transfer, the server directory structure will look like this:



We can transfer a different child subdirectory, as follows:

```
.\FastDataTransferClient.exe -n Upload -c Disk -s Disk -d d:\FdtClientRoot -r Recurse -f Foo\*
```

And after that the server structure will look like this:

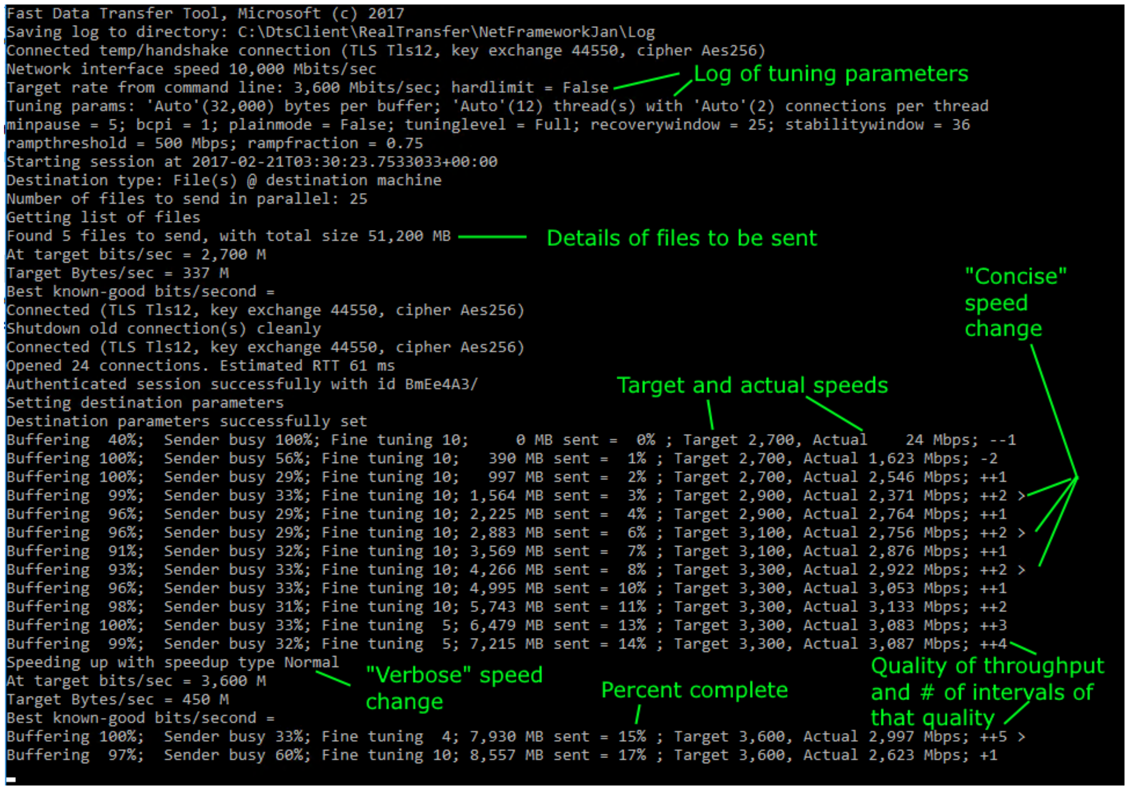


In summary, the pattern is that the client-side folder structure gets replicated onto the server, subject to any filter than we set on the client.

All of the above can be used with UNC paths. E.g. instead of D:\\FdtServerRoot you can use \\\\serverName\\rootShareName or even just \\\\serverName. (Note the doubling of the backslashes, in the server config file). In the latter case, a client-side filter like -f Bar* will map to the share \\serverName\Bar. FDT will not create the share, so it must already exist.

Understanding the tool's output

The sender-side output (i.e. the client for uploads) looks like this:



```
Fast Data Transfer Tool, Microsoft (c) 2017
Saving log to directory: C:\DtsClient\RealTransfer\NetFrameworkJan\Log
Connected temp/handshake connection (TLS Tls12, key exchange 44550, cipher Aes256)
Network interface speed 10,000 Mbits/sec
Target rate from command line: 3,600 Mbits/sec; hardlimit = False
Tuning params: 'Auto'(32,000) bytes per buffer; 'Auto'(12) thread(s) with 'Auto'(2) connections per thread
minpause = 5; bcpi = 1; plainmode = False; tuninglevel = Full; recoverywindow = 25; stabilitywindow = 36
rampthreshold = 500 Mbps; rampfraction = 0.75
Starting session at 2017-02-21T03:30:23.7533033+00:00
Destination type: File(s) @ destination machine
Number of files to send in parallel: 25
Getting list of files
Found 5 files to send, with total size 51,200 MB
At target bits/sec = 2,700 M
Target Bytes/sec = 337 M
Best known-good bits/second =
Connected (TLS Tls12, key exchange 44550, cipher Aes256)
Shutdown old connection(s) cleanly
Connected (TLS Tls12, key exchange 44550, cipher Aes256)
Opened 24 connections. Estimated RTT 61 ms
Authenticated session successfully with id BmEe4A3/
Setting destination parameters
Destination parameters successfully set
Buffering  40%;  Sender busy 100%; Fine tuning 10;     0 MB sent =  0% ; Target 2,700, Actual    24 Mbps; --1
Buffering 100%;  Sender busy 56%; Fine tuning 10;   390 MB sent =  1% ; Target 2,700, Actual 1,623 Mbps; -2
Buffering 100%;  Sender busy 29%; Fine tuning 10;   997 MB sent =  2% ; Target 2,700, Actual 2,546 Mbps; ++1
Buffering  99%;  Sender busy 33%; Fine tuning 10; 1,564 MB sent =  3% ; Target 2,900, Actual 2,371 Mbps; ++2 >
Buffering  96%;  Sender busy 29%; Fine tuning 10; 2,225 MB sent =  4% ; Target 2,900, Actual 2,764 Mbps; ++1
Buffering  96%;  Sender busy 29%; Fine tuning 10; 2,883 MB sent =  6% ; Target 3,100, Actual 2,756 Mbps; ++2 >
Buffering  91%;  Sender busy 32%; Fine tuning 10; 3,569 MB sent =  7% ; Target 3,100, Actual 2,876 Mbps; ++1
Buffering  93%;  Sender busy 33%; Fine tuning 10; 4,266 MB sent =  8% ; Target 3,300, Actual 2,922 Mbps; ++2 >
Buffering  96%;  Sender busy 33%; Fine tuning 10; 4,995 MB sent = 10% ; Target 3,300, Actual 3,053 Mbps; ++1
Buffering  98%;  Sender busy 31%; Fine tuning 10; 5,743 MB sent = 11% ; Target 3,300, Actual 3,133 Mbps; ++2
Buffering 100%;  Sender busy 33%; Fine tuning  5; 6,479 MB sent = 13% ; Target 3,300, Actual 3,083 Mbps; ++3
Buffering  99%;  Sender busy 32%; Fine tuning  5; 7,215 MB sent = 14% ; Target 3,300, Actual 3,087 Mbps; ++4
Speeding up with speedup type Normal
At target bits/sec = 3,600 M
Target Bytes/sec = 450 M
Best known-good bits/second =
Buffering 100%;  Sender busy 33%; Fine tuning  4; 7,930 MB sent = 15% ; Target 3,600, Actual 2,997 Mbps; ++5 >
Buffering  97%;  Sender busy 60%; Fine tuning 10; 8,557 MB sent = 17% ; Target 3,600, Actual 2,623 Mbps; +1
```

At the start, you will see the target speed quickly ramp up to the value specified by the -M parameter. After that, every so often, you will see notifications of automatic changes in target speed. For brevity, most speed changes are logged concisely, with just a ">" or "<" on the far right-hand side. But some are logged more verbosely, as shown in the screenshot above.

When the tool speeds up, it simply tries going faster on the existing connections. It will do this if it observes sufficient intervals of good quality throughput. On the other hand, it will slow down if it observes sufficient intervals of poor quality throughput. If the intervals are particularly poor, it will break all the existing TCP connections and create new ones.

Monitoring for success or failure

The client app will show, at the end of each transfer, whether it succeeded or failed. This information is shown in human-readable form on screen, and in the log file (if logging is enabled with the -L parameter). The program also returns a standard exit code, where 0 indicates success and non-zero indicates failure.

IMPORTANT: if there is an error, the client does not pause to wait for a key press. It just logs the failure to screen and log file (if enabled), sets a non-zero exit code, and exits. So you must not interpret "the program closed" as "success". This is particularly important if you run it from a Windows batch file, since it won't stay on screen after it closes – unless you add "pause" as the final line of your batch file.

Charting behavior

It can be helpful to chart the behavior of the app, for your own analysis of its performance. You can also supplement any log files you may send us with screenshots of the charts. On Windows, you can do this by using Perfmon.exe on the sending machine, and charting

- TCPv4 -> Segments Sent/sec (suggested "scale" setting: 0.0001 for networks faster than 1 Gbps, else 0.001)
- TCPv4 -> Segments Retransmitted/sec (suggested "scale" setting: 1.0)

On Linux you can capture the equivalent statistics in textual form, using tools such as sar and dstat.

Running the server interactively

If desired, you can run the server interactively instead of as a service/background job.

- Windows:
    1. First, stop the Service from the Services control panel app, or by running `FastDataTransferServer.exe stop`
    2. Then run the process interactively by running `FastDataTransferServer.exe run`
    3. To stop interactive execution, press CTRL-C.

4. To restart the service, do so from the Services control panel app, or by running `FastDataTransferServer.exe start`

- Linux:
  1. Stop the background process by running

     `sudo /opt/fdt-server/ShutdownInBackground.sh`.
  2. Run it interactively, *under the fdt user account*, with

     `sudo su fdt -c "/opt/fdt-server/FastDataTransferServer"`
  3. To stop interactive execution, press CTRL-C.
  4. To run it in the background again, run

     `sudo /opt/fdt-server/RunInBackground.sh`

FAQ

How does FDT ensure the integrity of my transfer?

FDT computes hashes at both the sending and recieving ends of the link. The sender computes a hash as it reads the file, and the reciever computes a hash as it writes the file. FDT only reports the transfer as successful if both hashes match. This verifies that what FDT wrote exactly matches what it read.

In FDT v1.0.0.x, the hash algorithm used is MD5.

FDT is slower than I expected. What can I do?

See the Tuning section below, and *please contact us* so that we can help, and so that we can learn how to make FDT go faster in environments like yours.

How is the FDT Authentication Key secured?

The security of the FDT authentication key when at rest (i.e. on disk) relies on Operating System Access Control (permissions) for the files that contain it. Those file-system permissions are the sole mechanism protecting the key on Linux. On Windows, in addition to file-system permissions, the key can also be encrypted using DPAPI. How can I encrypt the authentication key? for details.

Security of the FDT Authentication key in transit, i.e. when transmitted from client to server, is covered by the use of SSL/TLS - which is used for *all* traffic between FDT client and server.

If someone stole my FDT Authentication Key, what could they do?

If your FDT server is configured for disk access they could read and write all files in the configured "DiskLocation" directory.

If your FDT server is not configured for disk access, then the worst they could do is steal CPU and network capacity from your FDT server, to do FDT transfers to their *own* Storage account. They would not be able to access *your* Storage account by stealing the FDT Authentication Key alone.

Does FDT use *two* kinds of connection string?

Yes. They are:

- The "FDT Server Connection Strings". These are the strings that are output by the ARM template, and which you copied into the "ConnectionStrings" section of the client-side config file. They are documented in the next FAQ entry, below.
- The "Storage Connection Strings", which tell FDT how to connect to Azure Storage. They are documented above, in the section on Saving to Azure Storage.

What is the format of the FDT Server Connection strings?

An FDT Server Connection string can contain the following elements:

Name	**Required**	**Description**
ServerAddress	Y	Address of the server. Use a Fully Qualified Domain Name if you have one, otherwise an IP address.
Port	N	TCP Port number. Defaults to 10256.
ServerName	If, and only if, using self-signed cert	Name of server. Populate this only if ServerAddress is an IP address and you are using a self-signed SSL/TLS certificate. It says which server name FDT should expect to find in the certificate.
Thumbprint	If, and only if, using a self-signed cert	Thumbprint of the expected SSL/TLS certificate. Populate this only if you are using a self-signed SSL/TLS certificate.

What TCP port does FDT use?

The default is 10256. To override the default, you must do so both server-side and client side.

On the server:

1. Add a line of the form
```
"Port": 12345,
```

to FastDataTransferServer.settings.json.

2. In the Azure Portal, find the Network Security Group for the server(s), and change the port number in the rule named "allow-fdt"

3. If on Windows, open Windows Firewall and change the port number in the rule named "_allowFDT".

On the client: specify the port in the FDT Server Connection string (see FDT Server Connection Strings above).

How do I need to configure my firewall?

To use FDT, the client needs the ability to make outbound connections on the selected TCP port (see above for the port number). Nothing else is required at the client end.

At the server end, the server needs to be able to accept inbound connections on the selected port. When the ARM template creates servers, it sets up the necessary Network Security Group and Windows Firewall rules (for the default port number).

How does FDT recover from errors?

If there is a loss of connectivity, or other fatal error, the FDT client application will respond by waiting for a period of time (up to 30 seconds) and the starting a new transfer session to transfer only those files that were not sucessfully transferred in the previous attempt. FDT will keep retrying until either:

- All files are successfully transferred OR
- Five attempts in a row each fail without transferring even one single file.

This functionality is completely automatic and there are no parameters to control it.

Note: that in version 1.0.0.x the retry functionality is fully implemented only for *Uploads*. For *Downloads* it will retry, but when it does so it will process *all* files (including any that were already successfully transferred). This is a temporary limitation, which we plan to address in a future release.

Note also that error recovery is not available when you are using FDT's fake source or destination. I.e. when the command line contains `-c Null` or `-s Null`, since recovery is not meaningful in those cases.

Can I use multiple client machines at the same time?

We offer built-in support for scaling out over multiple servers (when uploading). There is no equivalent support for using multiple clients. The only way to use multiple client machines is for you to initiate a separate FDT transfer on each one. That works, as long as:

- The sets of files transferred by each client machine are disjoint - e.g. they come from separate directories, or match separate wild-card patterns. Do not use mulitple clients if more than one client may end up processing the same file.
- AND the total transfer throughput remains within your network capacity. E.g. if you are uploading over a 10G link, and your client-side disks are fast enough, one 16-core client can fill that link. So there would be no benefit in adding a second client. (If, on the other hand, your disks could not sustain 10 Gbps, you could run two clients with each uploading a separate set of files from separate disks. In that case you should set the -M parameter to half what you would use for a single client.)

Downloads are much slower than uploads. What can I do?

First, check the tuning advice below.

If download speeds are significantly slower than your upload speed to *one* server, then you can try changing network buffer sizes on the client machine. These changes *may* adversely impact the performance of other applications running on the same machine, so be prepared to reverse them if necessary. The changes are:

- On Windows: Note down the old values, then try increasing the send and recieve buffer sizes on your network adapter to their maximums. Also in your network adapter settings, ensure that Receive-side scaling (RSS) is enabled.
- On Linux: Note down the old values, then try increasing the TCP max buffer sizes (net.core.rmem_max and net.core.wmem_max) and the limits for TCP auto-tuning (net.ipv4.tcp_rmem and net.ipv4.tcp_wmem). See the server-side FDT Linux install script for the values we use of FDT servers. You can use the same values on your client.

Where are the server-side executables?

On Windows they are in c:\FDT\Server and on Linux they are in /opt/fdt-server.

Where are the server-side log files?

On Windows they are in c:\FDT\Server\Log and on Linux they are in /var/opt/fdt-server/logs

I always use the same client-side parameters. Can FDT automatically read them from a file?

Yes.

1. First, you need to find the "long name" for each of the parameters that you want to set. For instance the -c parameter has a long name of ClientDataLocation. See the parameter names listed in the section Finalize Client-Side Parameters or run the client with --help as the only parameter, and it will show you a full list of all parameters.
2. Once you know the long names of the parameters that you want to set, edit FastDataTransferClient.settings.json. Find the line for each parameter you want to set,

uncomment it by removing the leading // characters, and replace the existing placeholder value with the value that you want the parameter to have.

3. Save the file and run the client.

If any parameter is supplied in both the file and the command line, the command line value takes precedence.

How can I encrypt the authentication key?

Encryption of authentication keys is only supported on Windows, because Linux doesn't offer an equivalent of Window's Data Protection API (DPAPI).

On **Windows clients**, you can use user-scoped encryption. That means that the key can only be decrypted by the user who encrypted it. You can encrypt the key as follows:

1. First, find the AuthenticationKeyProtectionType line in FastDataTransferClient.settings.json. Remove the // characters from the start of it, and check that its value is "User".
2. Next, edit the first of these PowerShell commands to include your authentication key, then run the commands.

```
$bytes = [System.Text.Encoding]::Unicode.GetBytes("YourAuthKeyGoesHere")

Add-Type -AssemblyName System.Security

$protectedBytes = [System.Security.Cryptography.ProtectedData]::Protect( `
  $bytes, $null, [Security.Cryptography.DataProtectionScope]::CurrentUser)

$protectedString = [System.Convert]::ToBase64String($protectedBytes)

echo $protectedString
```

3. Finally take the output, which is quite a long string, and use it as the value of the AuthenticationKey in FastDataTransferClient.settings.json.

On **Windows servers** created with version FDT v1.0.0 or later, the FDT server setup script automatically applies machine-scoped DPAPI protection to the password. Machine-scoped protection means that software running on the FDT server machine, under any user account, can decrypt the key. The key cannot be decrypted on any other machine. (If you'd rather use user-scoped protection, see Can I run the FDT server under a different user account? because user-scoped protection is difficult under the default account.)

Can I encrypt the Storage Credentials?

Encryption of Storage Credentials is only supported on Windows, because Linux doesn't offer an equivalent of Window's Data Protection API (DPAPI).

To encrypt Storage Credentials:

1. Use the same PowerShell script as outlined above, under How can I encrypt the authentication key?. On the first line replace "YouAuthKeyGoesHere" with your Storage Account Key.
2. The output of the PowerShell script will be a long encrypted string.
3. Get your existing ServerStorage connection string in a text editor (e.g. Notepad), and replace the value of the key with the long encrypted version of it.
4. Add a new section to the ServerStorage connection string. The new section should be `AccountKeyProtectionType=User`.
5. Your new ServerStorage connection string will now be very long, probably too long to conveniently use on the command line (although you can do it that way if you wish). The easiest option is usually to omit it from the command line, and put it into the FastDataTransferClient.settings.json instead. To do so, remove the leading // characters from the ServerStorage line, and put your new long connection string on that line. When finished, the relevant portion of the file should look something like this:

```
//"ServerDataLocation": "Null",
"ServerStorage":

"AccountName=...;AccountKey=AQAAANC...==;AccountKeyProtectionType=User;ContainerName
=...",
//"TargetMegabitsPerSecond": 3000,
```

Can I use fewer servers?

These installation instructions suggest one server for every 2.5 Gbps of desired throughput. That's a good general-purpose rule of thumb. However, if you are saving to the VM's file system (rather than Azure Storage) and/or the network latency (RTT) between client and server is less than about 20 ms, you may be able to get higher traffic per server. To find out, simply take a backup of FastDataTransferClient.settings.json, then remove some of the connection strings from the original, save the file, and run some test transfers. If you're able to achieve the desired speed with fewer VMs, then you can simply delete the unneeded servers.

Can I use UNC paths?

Yes. Client-side directories and server-side directories may both use UNC paths. (Although, on the server-side, you may need to change the service to run under a different account in order to access them.)

Can I run the FDT server under a different user account?

Yes. By default FDT runs under "Local Service" on Windows and a user named "fdt" on Linux. On Windows you can use the Services UI to change the account that the service runs as, and on Linux you can edit the user account in /opt/fdt-server/RunInBackground.sh. Then you **must** set appropriate permissions on the directory where the binaries reside so that your new account can run FDT; and on the log directory so it can create, delete and write to log files; and on the SSL/TLS certificate. To set the certificate permissions:

- On Windows: right-click the certificate in "Manage Computer Certificates", choose All Tasks -> Manage Private Keys; and give Read access to the new account and revoke it from Local Service.
- On Linux: move the fdt user's .dotnet directory to the home directory of your new user, and then
  chown -R <newUsr>: /home/<newUsr>/.dotnet/corefx/cryptography/x509stores/my

Finally, if running on Windows, you might like to consider changing the DPAPI protection that is applied to the AuthenticationKey in FastDataTransferServer.settings.json, from "Machine" scope to "User" since "User" is theoretically even more secure. Details are beyond the scope of this documentation, but if you are familiar with DPAPI for service accounts you may re-encrypt the authentication key with "CurrentUser" scope. Put the new encrypted value into the server's settings.json and change AuthenticationKeyProtectionType in that file from "Machine" to "User".

How do the filter wildcards work?

When reading from Storage, the only wildcard allowed is a simple trailing *.

When reading from disk, but not using the Recurse option, you can use a * to represent the unknown portion of a filename. E.g. *.txt for all text files on Windows; foo* for all files with names starting with foo. When *not* using Recurse only the root directory nominated by -d is searched. No subdirectories are examined.

When reading from disk with -r is set to Recurse, here are some examples to show how the filter wildcards work:

- A simple filename finds all files with that name in all subdirectories. E.g. foo.txt will find files named "foo.txt" in all directories - including the root directory and all its subdirectories (recursively).
- A simple filename with a wildcard such as foo* finds all files starting with "foo" in all directories - including the root directory and all its subdirectories (recursively).
- A simple path wildcard such as bar*, where "bar" is a subdirectory of the root, will find all files in the "bar" directory *and in "bar's" subdirectories* (recursively). **PERFORMANCE NOTE** This is the only type of wildcard for which FDT 1.0.0.5 can use its fast parallel algorithm to scan the file system. For all other types of wildcard, it will

fall back to the slower sequential scan. You'll usually only notice the difference if you are transferring at least a million files.

- A single-star path wildcard such as foo\bar*\aaa* finds intermediate directories that start with "bar". E.g. it will match files under "foo\barrier\aaa" and "foo\barrel\aaa" but not "foo\boo\aaa". It will NOT match "foo\barrier\somethingelse\aaa" because it only matches *one* directory level.

- A double-star path wildcard stands for zero or more directory levels. So foo**\bar* would match "foo\bar" and "foo\aaa\bbb\bar" but not "foo\aaa\bbb\ccc" (since the latter does not include a directory named "bar"). As in the examples above, the trailing single "*" is to match the *files* inside each matched directory.

- **NOTE** A path with no wildcards acts as if a ** had been inserted before the last directory separator. For instance aaa\bbb\ccc\foo.txt will act like aaa\bbb\ccc\**\foo.txt. It will find "foo.txt" in aaa\bbb\ccc *and* recursively in any subdirectories of aaa\bbb\ccc. The only way to prevent this behaviour, and to focus only on a single directory without any recursion, is to remove the -r Recurse parameter and set the root directory (-d) to be exactly the specific directory that you want to process.

Note to Windows users: ending the filter patten with * is exactly the same as ending it with *.*. Both will find *all* files, with and without extensions.

Tip: when using an unfamiliar filter wildcard, you can use --LogCompletedFiles Always to get a log of exactly which files were transfered. That will allow you to check whether the behaviour was what you expected.

What if I want to specify particular files to transfer?

You can nominate particular files to transfer, instead of just using a wildcard pattern. Add the --FileNames parameter to the client side command line, and follow it by a *space delimited* list of files to transfer. E.g.

```
FastDataTransferClient.exe -n Upload -c Disk -s Disk -d c:\foo --FileNames "c:\foo\file1.dat" "c:\foo\bar\file2.dat"
```

Note that the -d parameter must still be provided, to avoid ambiguity the filenames must have absolute (not relative) paths, and each file must be somewhere inside the root folder nominated by -d.

Alternatively, you can list the files in a text file, and identify the text file with the --FileNamesFile parameter.

We recommend that you **do NOT** use --FileNames, or --FileNamesFile, at the same time as --RunDateLogFile.

What if I want to rename files during the transfer?

You can do this, but only when using the `--FileNames` or `--FileNamesFile` parameters as above. Simply follow each filename with the pipe (|) character, then the name you want it to have at the destination. Directories in the destination name become directories at the destination ("virtual" directories if the destination is Storage). E.g.

```
FastDataTransferClient.exe -n Upload -c Disk -s Disk -d c:\foo --FileNames
"c:\foo\file1.dat|somedir/destname1" "c:\foo\bar\file2.dat|destname2"
```

When including directory separators in the desired name, you can use either forward or backslashes. FDT will automatically convert them to the right format for the destination.

What about empty directories?

FDT only transfers files. Because an empty directory contains no files, empty directories are not transferred. They are, however, listed in a log file. To get the names of the empty directories from the log file:

1. Check the start of the main log file, to make sure it says `Using parallel file search with x threads`. Empty directories can only be logged when parallel file search is used. If FDT has to fall back to a sequential search, there will be a message to that effect near the start of the main log, and no empty directories will be logged. FDT falls back to sequential search if you specify a complex filter.

2. After the transfer completes, search the "error files" log for lines containing "Empty directory skipped". The log you need will be in the same directory as the main log file, but it will have a name of the form "error-files-yyyymmdd.log". Note that an entry in that file does not necessarily signal an error (some are warnings or merely informational) and only those lines that contain the text "Empty directory skipped" signal a skipped directory.

What punctuation should be in my certificate thumbprints?

Some tools punctuate certificate thumbprints with spaces, others with colons, and others don't punctuate them at all. All formats are fine for FDT. The FDT client and server ignore spaces and punctuation in thumbprints.

How can I use FDT with a clustered file system?

If your Azure Linux VMs are running a clustered file system, such as Gluster or BeeGFS, you can use FDT to load data directly to the clustered file system. One possible configuration is to install FDT on a subset of your cluster nodes. For instance, in testing we achieved a data transfer speed of 9.5 Gbps to a six-node BeeGFS cluster, with just two of the nodes also running the FDT server software. After creating your cluster, you can install FDT on some on the nodes by following the instructions for Installing FDT on an existing server. Or, if you prefer to run FDT on VMs that are separate from those running your cluster, you can install new FDT VMs with the usual FDT server installation template.

Can the FDT server write to Azure Files?

Yes. Connect the file system of your FDT server to the Azure Files share (i.e. with a mapped drive on Windows, or a mount point on Linux). Then send data to the share by following the instructions for using server-side disk. As at October 2018, this is the only supported way to use FDT with Azure Files.

Can the FDT client read from Azure Storage?

Having the client read from Storage is useful when transferring data between Azure regions. The client reads from its local region and the server, in another region, writes to *its* local Storage.

This feature needs to be manually enabled, because the client does not ship with storage connectivity enabled. To enable and use this feature:

1. NOTE: ONLY perform these steps if your FDT client installation is up-to-date (on the latest version). Do not perform these steps if, for some reason, you are deliberately using an older client version (because these steps will then have the unexpected side effect of updating your client). You can see which version you have at the start of the client-side output or log file, and you can see which is the latest available version here.

2. Make a backup of FastDataTransfer.dll from your FDT client directory. (In the unlikely event of needing to roll back the change, simply copy this file back to the client directory.)

3. Place server-side executable and dll files into the *client's* directory as follows:

   – On Windows, download https://aka.ms/fast-data-transfer-windows-server-storage-support. Unzip the file, and put its contents into your FDT client directory. It is safe (but not necessary) to overwrite any existing files that have the same names. When you are finished, FastDataTransferClient.exe and FastDataTransferServer.exe should be in the same directory on the client side.

   – On Linux, download https://aka.ms/fast-data-transfer-linux-server-storage-support, un-tar its contents, and put them into your FDT client directory. It is safe (but not necessary) to overwrite any existing files that have the same names. When you are finished, FastDataTransferClient and FastDataTransferServer should be in the same directory on the client side.

4. Set the -c parameter to Storage on the client side. This tells the *client* to access Storage. The client will load the necessary logic from the files you copied.

5. Add a --ClientStorage parameter to the client-side command line. The format is the same as that documented for the --ServerStorage parameter, but this time it specifies which Storage container the *client* should connect to.

6. (Optional) Optimize the client machine's network adapter and OS settings to maximize the speed at which is can retrieve data from Storage. To do this, follow the same network tuning advice given above, under Downloading.

7. Run the client.

What data does FDT collect?

FDT collects:

1. The main log files, which are saved only to your local file system. You may manually send them to us if requesting support. These files have names beginning with "log-".

2. Log files that record the names of files that have errors - e.g. which cannot be read. These are stored in the same directory as the main log files, but they have names beginning with "error-files".

3. Auxilliary log files that record the names of all successfully transferred files. These are stored in the same directory as the main log files, but they have names beginning with "completed-files". These are only created if you enable them through the LogCompletedFiles parameter. For privacy reasons, please do not send us these files when requesting support.

4. Telemetry (anonymous performance statistics). These statistics are sent automatically to Microsoft, to assist with on-going development of FDT, unless you change TelemetryType in FastDataTransferClient.settings.json to None. If you want to see the data that is sent, you can instead set the value to SendAndLog which will send the telemetry and will also include all the sent data at the end of the client-side log file.

Does FDT automatically update itself?

No. But this release does *tell* you when a new update is available. If an update is available the following message will appear near the start of the client-side log/output, and will be repeated again at the end:

```
*************************************************************
A new version of Fast Data Transfer is available. See http://aka.ms/fast-data-transfer-update-info
*************************************************************
```

If you manually browse to the URL supplied in the message, you'll find instructions on how to upgrade to the new release.

If you want to disable the version check, you can do so by setting NotifyOfUpdates to false in FastDataTransferClient.settings.json.

Note: if you would like to script an automatic check for new versions, you can use the --TestUpToDate command-line parameter. When that parameter is supplied, the client application *only* checks for updates; no data transfer is performed. No other command line parameters are required when using --TestUpToDate. Unlike an actual data transfer, in this case the process exit code will be set to 0 (success) if the application is still up to date, and 1 (failure) if it is out of date. (The exit code is also 1 (failure) if the version data file could not be fetched over the internet).

Where can I get help?

Email FastDataTransferHelp@microsoft.com or visit http://aka.ms/fast-data-transfer-feedback to contact us about problems, make suggestions for improvements, or just tell us how Fast Data Transfer is working for you.

Tuning

This section contains guidelines on diagnosing and fixing performance problems.

First, ensure the target speed (-M on the client-side command line) is set something realistic for your environment, as described above. This makes for a smoother first few minutes of the transfer, because the tool does not have to spend so much time "hunting" for the right value. Also, at the end of *every* transfer, you can read the final log file lines (on the sending side) where FDT will include any suggested changes to the target for future runs.

If you're still not getting the throughput you expect, we suggest you try the following rules of thumb:

1. **Are the sender-side disks too slow?** Check "*Buffering %*" in the sender-side log. Ideally it should be consistently near 90% or above. If it is often under 5% that indicates your sender-side disks are slower than your network. In that case you can:

   - Try changing NumberOfParallelFiles in the client side config file. If you are reading from physical spinning disks, you should try setting this value to 1. That makes FDT read just one file at a time, and substantially reduces the overhead of disk seek times. If you are reading from SSDs or a SAN, you can instead try *increasing* NumberOfParallelFiles (e.g. to 64).

   - For all types of file system, you can also experiment with increasing the DiskReadBufferKilobytes parameter to 128 or 1024.

   For these config changes, remember to remove the // comment markers from the start of the relevant config line, and to check the start of the log to see if your change took effect. If these changes don't help, you may want to investigate striped disks (e.g. RAID or Windows Storage Spaces) or other alternatives to allow faster disk reads. Alternatively, you may consider using multiple client machines as described in the FAQ.

2. **Are the receiver-side disks (or Storage) too slow?** There are two ways to check this. One is to watch out of messages in the sender-side log, saying *"Receiver RAM usage is high. So delaying sending remaining files"*. The other is to check the "**MB in RAM"** figures in the receiver-side log. If you see the sender-side messages, or if the reciever side log shows that the RAM usage climbs to several gigabytes and stays

there, that suggests your final destination (disk or Storage) is slower than your network.

If saving to disk, then you can investigate striped disks or clustered file systems.

If saving to Storage, you can look at the IOPS and Retry % figures which appear in the reciever-side log when saving to Storage. A retry %age above about 2 or 3 percent shows that Storage is throttling your usage. IOPS numbers approaching the documented IOPS limits for Storage show that your number of operations per second may be the limiting factor. (The latter generally only happens with very small files). If the IOPS figure is a few thousand or less, and your transfer includes more than 16 files, you can try transferring a larger number of files in parallel. To do this, edit the FastDataTransferClient.settings.json, uncomment NumberOfParallel files, and set it to 32, 64 or something higher.

IMPORTANT: Note that Storage uses range-based partitioning to scale and load balance. Each blob has a key, composed of *container name + blob name*, and blobs with alphabetically similar keys are assigned to the same partition. If you have a lot of activity concentrated in a number of blobs whose keys (*container name + blob name*) start with the same characters, that will put a lot of load on the partition. Storage will then automatically split that partition, so that the load is spread across multiple servers. For example, if you upload to a newly-created container, all the blobs will initially be served by a single partition server because they are all in the same part of the key space (the part begining with your new *container name*). As Storage observes high load there, it will repartition that part of your key space across multiple servers, and throughput will increase. You can observe this with FDT by running a series of transfers to a newly-created container, using `-c Null --FakeSourceGB 50`. **Over the course of several transfers to the same container, the retry percentage will drop and throughput will increase.** Once re-partitioning is complete, FDT's throughput to Storage should approximately equal its throughput to "Null" (`-s Null`), provided each transfer contains blobs with alphabetically dissimilar names.

If you find that partitioning delays are significant in your case you can try changing the `FileOrder` setting in client settings file. Remove the leading // to uncomment the line, and then change the value from `BiggestFirst` to `Hybrid`. That may help in some cases.

If any problems persist, please contact us to discuss.

3. **Check 'Busy %' in the FDT client-side log**. If Busy % is consistently near 100%, and throughput is much, much lower than expected, you can try using a much higher connection count. This is a rare problem, which affects just a handful of FDT users with particular network conditions and hardware. If it affects you, try changing `NumberOfThreads` in the client side config file to 128, 256, 512 or 768. (Remember to remove the leading // from the line, to uncomment it.) You may need to experiment to see which of those figures gives best results. If none works, and Busy % remains high, please contact us for assistance.

4. **Is the sending machine under-powered?** Check for high CPU usage on the sending machine. If CPU usage is near 100%, then you may benefit from

    1. Using a sending machine with more CPU power, OR

    2. Increasing buffer size (see below)

5. **Are you limited by your network bandwidth?** Use a tool such as NTTTCP or iPerf3 to see how fast they can transfer data between your client machine and one of your FDT servers. That gives a good indication of the available network bandwidth, which is the maximum speed obtainable by *any* data transfer tool. FDT should come close to that maximum speed. (Although, unlike the testing tools, FDT may need to use multiple servers to do so.)

    When using NTTTCP or iPerf, remember to (temporarily) allow its ports through the VM's Network Security Group (in the Azure Portal). On Windows, also remember to allow the program through Windows Firewall.

    If you use these tools, and find that your available network bandwidth is a problem, you may like to investigate upgrading your network capacity. Microsoft ExpressRoute offers a selection of high-speed network connections direct to Azure.

6. **Is your network provider throttling you at a particular level?** Check the sender-side log. If it shows that actual throughput always collapses as soon as it gets above one specific level, you can prevent the tool from even trying speeds above that level. Simply look at the log to find the highest “target rate” figures where throughput is stable, use the figure as the value of the -M parameter, and in FastDataTransferClient.settings.json, uncomment "SpeedLimit" and set its value to "Fixed". That will prevent the tool from ever trying any higher target rate.

7. **Is a proxy, or similar, slowing your traffic?** If the Busy % in the FDT client log is 100%, or very close to 100%, and yet the throughput is very low (e.g. 10 Mbps) this can be sign that something in your network infrastructure is slowing your traffic - e.g. a proxy or security device that inspects your outbound traffic. Talk with your network team to find out if this might be the case.

8. **Can FDT's buffer size be optimized?** Sometimes it helps to adjust the size of the units in which FDT sends data.

    – *Should you increase the buffer size?* If throughput is very stable (i.e. actual speeds reported throughout the transfer have very few big drops) then you can try increasing the buffer size. Look near the start of a sender-side log file to see the default buffer size that the tool has selected for your transfers. The value is typically between 4000 and 48000 bytes. Double that number, and use it as the value of MaxBufferSize in FastDataTransferClient.settings.json. (Remember to remove the leading slashes, to uncomment the relevant line of the file.) If test runs are still stable, try doubling and retesting a few more times. Stop when either (a) the increase does not improve the max speed reached or (b) instability sets in (in which case you should drop back to a slightly smaller, more stable, number).

  – *Should you decrease it?* If throughput keeps dropping off suddenly, but its not at any one specific threshold speed, it can sometimes help to decrease the FDT buffer size. To do this, follow the same process as for increasing, but make the size smaller instead of larger.

9. If none of the above works, contact us, and ideally send us some log files to illustrate how the tool is behaving in your environment.

Uploading or downloading a few large files

Azure Storage and some disk file systems are faster when processing many files in parallel, rather than just one file at a time. If your data transfer contains just a small number of big files (fewer than about 16 files in total) you may see reduced performance from disks and/or Storage. We are aware of this problem and hope to address it, as far as possible, in future releases.

Uploading or downloading lots of small files

FDT's client-to-server protocol is efficient for all file sizes. However, when transferring lots of small files – files less than about 50 KB in size – performance can suffer in the link between FTD and Disk/Storage. This applies to both uploads and downloads.

The following points may help to improve performance when transferring small files:

1. For uploads, you can use a larger number of servers.
2. For both uploads and downloads, FDT's throughput to Azure Storage tends to be constrained by the Server VM's CPU capacity. You can try a larger VM size for the VMs that communicate with Storage.
3. For uploads, check that FDT reports that it is using a "parallel" scan to scan the file system at the start of the transfer. (This is the default, but with some complex filter criteria, FDT will fall back to a sequential scan. You can see which it has selected in the log, at the point where it looks for the files to transfer.)
4. In you have a very large number of files in your transfer, e.g. 10's of millions, make sure your client machine has at least 16 GB of RAM, preferrably 32 GB. And monitor RAM usage as FDT runs.
5. Contact us if you still have problems. We can help you with tuning its disk reads from your source file system, and may also be able to help with other issues related to transfering small files.

Upgrading

These instructions are for upgrading from version 1.0.0.x to a later version. E.g. from the July 2018 release to the October 2018 release. If you have an older installation, prior to May 2018, we recommend to start afresh rather than updating it.

Upgrading the client:

1. Make a new, empty, directory to contain the new version.
2. Install the FDT client into the brand-new directory, by following the instructions at the start of this document. **BUT** don't bother manually editing the config file, because you can re-use your old one. (See next step.)
3. From the directory containing the **old** version, copy FastDataTransferClient.settings.json into the **new** directory. I.e. copy your known good settings over top of the default settings file that was included in the new installation.
4. Check the file permissions on the copied file, to make sure they are just as strict as the permissions on the original (because it contains an authentication key.)
5. Consider copying the contents of the old version's "Logs" directory into the new one, so that you won't lose any log files if, or when, you delete the old version.
6. Delete the old version when you are happy with the new one

Upgrading the server:

(Note: if your server does not hold any data that you need to keep, and if you created it with the supplied FDT template, the easiest way to "upgrade" is often to just throw it away and make a new one with with the template. The new one will have a different IP address and connection string, which you'll need to put into the client config. But if your server stores your data on its own drives, or if you simply want to keep the existing VM, read on for instructions on how to manually upgrade it.)

To manually upgrade an existing FDT server:

1. Check and note down the file permissions that are set on FastDataTransferServer.settings.json, which you can find in the FDT server install directory.
2. Take a backup of FastDataTransferServer.settings.json
3. Stop the FDT service. In Windows this means stopping the Fast Data Transfer Service, and in Linux it means using the shell script provided for stopping the server:
   ```
   sudo /opt/fdt-server/ShutdownInBackground.sh
   ```
4. On Windows, make a backup of the Logs subdirectory under the Fast Data Transfer server directory, if you want to keep old logs. (On Linux, logs are saved elsewhere, so they won't be affected.)

5. Delete the contents of your Fast Data Transfer server directory

6. Download aka.ms/fast-data-transfer-windows-server on Windows or aka.ms/fast-data-transfer-linux-server on Linux and extract its contents into your Fast Data Transfer server directory. (The same directory as you used before).

7. Copy the backup of FastDataTransferServer.settings.json back into that directory and then *delete your backup* of that file (since it contains an authentication key).

8. Check that FastDataTransferServer.settings.json, which you just copied back, still has the same permissions that you noted down at the start. (Important since it contains an authentication key.)

9. On Windows, if you haven't done so already, consider encrypting the Authentication Key using the new DPAPI encryption support. See How can I encrypt the authentication key?.

10. Restart the Fast Data Transfer service (Windows) or `sudo /opt/fdt-server/RunInBackground.sh` (Linux)

Installation of Linux dependencies

To run FDT on Linux, you need to install the Linux packages that .NET Core depends on. When installing Linux servers using the supplied template, this is automatically taken care of your you. (The template uses "option 3" below). But when you install the FDT client on Linux, or when you install the server without using the template, you need to get the necessary files onto the target machine yourself. There are several different ways to do this:

Option 1: Install selected packages with package manager

In this option, you install only those packages which are not already present by default. This is a quick option and suits machines with internet connectivity:

- Fedora and CentOS: `sudo yum install libunwind libicu`
- SUSE Linux and OpenSUSE: `sudo zypper install libunwind libicu`
- Ubuntu and Debian: `sudo apt-get install libcurl3 libunwind8`

Option 2: Install the .NET SDK

This is a more comprehensive option, which actually installs more than FDT needs. Like the first option, this one requires an internet connection to download packages. Instructions are here.

Option 3: Install pre-prepared dependency packages

This option works on machines where firewalls prevent direct internet access.

We have prepared tarballs containing all the required files, and tested them on all the distros supported by the FDT server template. To use: 1. Create a directory called netcoredeps inside your FDT directory. (I.e. netcoredeps should be a sibling of FastDataTransferClient or FastDataTransfer server.) 2. Then get the relevant tarball, from the list below, and unpack it into the netcoredeps directory. For machines without direct internet access, you can download the tarball on a different machine, then move it to the target machine using SCP or similar.

When FDT runs, the files in the netcoredeps directory will automatically be found and used. Technical details of this approach are documented here.

List of tarballs

CentOS 7.2:

https://fdtreleases.blob.core.windows.net/linux-dependencies/netcoredeps-centos-7-2.tar.gz

CentOS 7.4:

https://fdtreleases.blob.core.windows.net/linux-dependencies/netcoredeps-centos-7-4.tar.gz

Debian 9:

https://fdtreleases.blob.core.windows.net/linux-dependencies/netcoredeps-debian-9.tar.gz

OpenSUSE Leap 42.3:

https://fdtreleases.blob.core.windows.net/linux-dependencies/netcoredeps-opensuse-42-3.tar.gz (note the dash between the 42 and the 3 if copying and pasting this)

Ubuntu 14.04:

https://fdtreleases.blob.core.windows.net/linux-dependencies/netcoredeps-ubuntu-14-04.tar.gz

Ubuntu 16.04:

https://fdtreleases.blob.core.windows.net/linux-dependencies/netcoredeps-ubuntu-16-04.tar.gz

Ubuntu 17.10

https://fdtreleases.blob.core.windows.net/linux-dependencies/netcoredeps-ubuntu-17-10.tar.gz

Option 4: Create your own tarball

This is the same as option 3, but allows you to create our own tarball for additional distros. We provide a script to create the tarball. You can download the script from https://aka.ms/fast-data-transfer-find-linux-dependencies. You will need a "source" machine where you can run the script. On that machine, you will need to be able to install packages using your distro's package manager. However, after you have made the tarball, you can take it to a target machine and you don't have to do anything on the target other than unpack the tarball as described in option 3.

Instructions for running the script are included in the script itself, as comments.

Installation on existing server VMs

Wherever possible, we recommend that you run the FDT server program on VMs created by the template that we supply. It's quick to set up and, also importantly, it offers you the easiest way to updgrade to new FDT versions - just discard the old VMs and run the template again.

However, if you need to run your FDT servers on existing machines, not created by the template, here's how to do it:

Windows

These instructions will install FDT into c:, and set it up as a Windows Service. They will set up a Windows Firewall rule, and create a self-signed SSL certificate for FDT to use. They will put the FDT authentication key into the FDT server's settings file, and restrict access to that file to Admins and Local Service (since that what the FDT service runs as, by default).

You can read the PowerShell script for more details.

1. Check the size of the server you plan to install onto. For 2+ Gbps of traffic to Blob Storage, a server with 16 CPU cores is recommended. If operating a lower speeds, or not saving to Blob Storage, it's OK to use fewer cores. We don't recommend you go below 4 cores. By default, FDT is configured to use up to 8 GB of RAM per session. (If your machine has 8 GB or less, in total, you should set MaxGigabytesPerSession in the Server-side settings.json file to 2 or 4.)

2. The server must be able to listen on the port used by FDT. That port is 10256 by default. For a typical Azure server configuration, this means you should find the Network Security Group that's associated with the server, and manually add a rule to that group to allow TCP traffic through on the port that FDT uses.

3. Ensure the server has the right version of the .NET Framework, i.e. version 4.6.2 or later. The easiest way to ensure that this is installed is to just to attempt to install it, and it will tell you if it, or a later version, is already present, in which case you can cancel the installer as soon as it shows that message. To attempt to install it, get the installer from here. If you'd rather not attempt the install unless absolutely necessary, you can follow the instructions here to check which version is currently present.

4. Create temporary directory (e.g. c:\temp).

5. Open a Powershell command prompt in that directory.

6. Run this command Get-ExecutionPolicy. If it returns "Restricted" or "AllSigned", then run this command: Set-ExecutionPolicy -Scope Process -ExecutionPolicy RemoteSigned

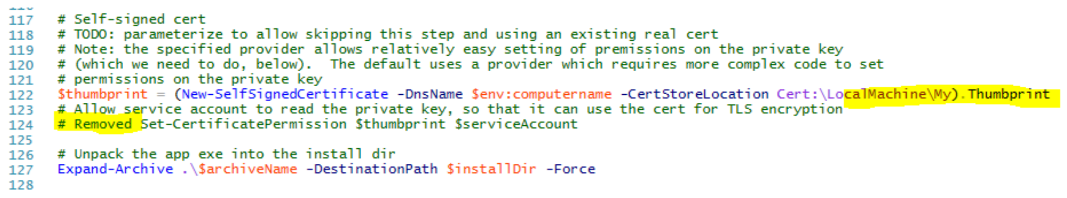
7. Download the install script by running this Powershell command

```
wget https://aka.ms/fast-data-transfer-setup-windows-server `
 -Out FDT-setup-windows-server.ps1
```

8. If running on a version of Windows prior to Server 2016 (e.g. Server 2012 R2)

   – Make sure PowerShell v5 is installed on it, since the install script requires PowerShell 5 or later, but its not installed on older versions of Windows by default. See here for install instructions.

   – Edit the downloaded install script. First edit the New-SelfSignedCertificate line to remove the -FriendlyName and -Provider paramemters. Then REMOVE the call to Set-CertificatePermission. The edited lines should look like this (changes highlighted):



```
# Self-signed cert
# TODO: parameterize to allow skipping this step and using an existing real cert
# Note: the specified provider allows relatively easy setting of premissions on the private key
# (which we need to do, below).  The default uses a provider which requires more complex code to set
# permissions on the private key
$thumbprint = (New-SelfSignedCertificate -DnsName $env:computername -CertStoreLocation Cert:\LocalMachine\My).Thumbprint
# Allow service account to read the private key, so that it can use the cert for TLS encryption
# Removed Set-CertificatePermission $thumbprint $serviceAccount

# Unpack the app exe into the install dir
Expand-Archive .\$archiveName -DestinationPath $installDir -Force

```

9. Download the zipped binaries into the same directory, by running this command

```
wget https://aka.ms/fast-data-transfer-windows-server -Out FDT-Windows-Server.zip
```

10. Create a strong password-like string of your choice. It must be at least 16 characters long, and must include letters, numbers and at least one special character (e.g. *#%"). This will be the FDT authentication key for your server. (If you are installing multiple servers, to share load across them, use the same authentication key for all of them). You will need to remember this, to put it into your client config file.

11. Edit this this command, to put your selected authentication key into it, then run it to convert your chosen authentication key into the required base64 encoding. (Base 64 encoding is used to work around problems with special characters, when the ARM template calls this script. You're not using the ARM template, but the script still requires the password to be base64 encoded).

```
$base64AuthKey=[System.Convert]::ToBase64String([System.Text.Encoding]::ASCII.GetBytes('yourFDTAuthKeyGoesHere'))
```

12. Then, from Powershell, run this command to install FDT: (NOTE: Change the final "true" to "false" if your VM will be reached over an ExpressRoute connection on its *private* IP address.) Don't worry if you briefly lose your remote desktop connection to the VM while this runs. That's normal and the connection will automatically be restored after a few seconds.

```
.\FDT-setup-windows-server.ps1 -archiveName FDT-Windows-Server.zip -authKey
$base64AuthKey -returnPublicIp true
```

13. Once the command completes, the last line of its output should be a string of the form: "a|ServerAddress=;ServerName=;Thumbprint=|a"
14. Strip off the leading and trailing "a|" and the trailing "|a", then record the remaining string for use as the FDT server connection string in your client side FDT config file.
15. Close your Powershell command window (to remove $base64AuthKey from memory)
16. If running on a version of Windows prior to Server 2016 (e.g. Server 2012 R2)
    - Manually give "Local Service" read access to the certificate's private key. Choose "Manage Computer Certificates" (under Control Panel) and find the certificate under "Certificates – Local Computer" -> "Personal" -> "Certificates". The certificate you are looking for will have the machine name in both the "Issued to" and "Issued by". Right-click the certificate, choose All Tasks -> Manage Private Keys; and give Read access to the account that FDT runs under - which is the 'Local Service' account by default. (If you need to double-check that you have the right certificate, compare its thumbprint to that supplied in the output of the FDT installation script.)
    - Restart the Fast Data Transfer service.
17. Go to your FDT client machine, put the connection string from above into its config file, also put the authentication key into the config file, and run a test.
18. High bandwidth users only: If you will be running with multiple FDT servers to scale out, and connecting to them from just ONE client, then repeat the above on the remaining servers, using the same authentication key each time. In the connection string section of the client config file you should end up with a list of connection strings, one for each server.

Linux

These instructions will create a user named fdt and start FDT running under that user account. Note that in this release, FDT on Linux does not automatically restart after reboots. (Although you can configure it as a systemd service yourself, if desired.) These instructions create a self-signed SSL certificate for FDT to use. They will put the FDT authentication key into the FDT server's settings file, and restrict access to that file to the "fdt" user.

You can read the script for more details.

1. Check the size of the server you plan to install onto. For 2+ Gbps of traffic to Blob Storage, a server with 16 CPU cores is recommended. If operating a lower speeds, or not saving to Blob Storage, it's OK to use fewer cores. We don't recommend you go below 4 cores. By default, FDT is configured to use up to 8 GB of RAM per session. (If

your machine has 8 GB or less, in total, you should set MaxGigabytesPerSession in the Server-side settings.json file to 2 or 4.)

2. Decide which option you will use for installing the packages that FDT depends on. See Appendix: Installation of Linux dependencies above. You will use this decision below.

3. If the server is in Azure and is in an Azure Network Security Group, then manually add a rule to that group to allow TCP traffic through (from your desired client machine(s). To do so, simply open inbound TCP traffic on the port that FDT uses. That's port 10256 by default.

4. Download the install script:

```
wget https://aka.ms/fast-data-transfer-setup-linux-server -O FDT-setup-linux-server.sh

chmod 700 FDT-setup-linux-server.sh
```

5. Download the FDT tarball into the same directory, by running this command:
`wget https://aka.ms/fast-data-transfer-linux-server -O FDT-linux-server.tar.gz`

6. If you decided to use the "tarball option" for installing the packages that FDT depends on, then download that tarball also, and put it in the same directory as the two files you have already downloaded. Your directory will now contain the following files FDT-setup-linux-server.sh, FDT-linux-server.tar.gz and netcoredeps-<something>.tar.gz

7. Create a strong password-like string of your choice. It must be at least 16 characters long, and must include letters, numbers and at least one special character (e.g. *#%"). This will be the FDT authentication key for your server. (If you are installing multiple servers, to share load across them, use the same authentication key for all of them). Remember this, because you will need it on the client side.

8. Edit this this command, to put your selected key into it, then run it to convert your chosen authentication key into the required base64 encoding. (Base 64 encoding is used to work around problems with special characters, when the ARM template calls this script. You're not using the ARM template, but the script still requires the password to be base64 encoded).

```
base64AuthKey=$(base64 <<< 'yourKeyGoesHere')
```

9. Now, for security, you should delete the previous command from your shell history (since it has a plaintext authentication key in it). On Bash you can do so by running this command to delete the immediately previous command from history:

```
history -d $(history | tail -n 2 | head -n 1 | awk '{print $1}')
```

10. Then run a command like the following to install FDT. Change the "true" to "false" if your VM will be reached over an ExpressRoute connection on its *private* IP address. Note that there are two options for the final parameter. If you are using a tarball for dependencies the final parameter must be the name of the tarball; otherwise it must

be the word 'MANUAL' to indicate to the script that you are manually deploying dependencies and it does not need to process a dependencies tarball. So *for example*, your command would look like this if using the Ubuntu 16.04 dependencies tarball:

```
sudo ./FDT-setup-linux-server.sh -archiveName FDT-linux-server.tar.gz -authKey
$base64AuthKey -returnPublicIp true -deps netcoredeps-ubuntu-16-04.tar.gz
```

and like this if deploying dependencies manually:

```
sudo ./FDT-setup-linux-server.sh -archiveName FDT-linux-server.tar.gz -authKey
$base64AuthKey -returnPublicIp true -deps MANUAL
```

11. Once the command completes, the last line of its output should be a string of the form:
    `"a|ServerAddress=<ipAddress>;ServerName=<serverName>;Thumbprint=<thumbprint>|a"`

12. Strip off the leading and trailing “a|” and the trailing “|a”, then record the remaining string for use as the FDT server connection string in your client side FDT config file.

13. Remove $based64AuthKey from memory by closing your shell session or running
    `unset base64AuthKey`

14. Go to your FDT client machine, put the connection string from above into its config file, and run a test.

15. High bandwidth users only: If you will be running with multiple FDT servers to scale out, and connecting to them from just ONE client, then repeat the above on the remaining servers, using the same authentication key each time. In the connection string section of the client config file you should end up with a list of connection strings, one for each server.

END