# ctsTraffic Usage

ctsTraffic is a tool built by the Windows Networking team originally designed at the start of Windows 8. It was built from the ground-up with core requirement to be able to shape and model API and protocol behaviors to verify synthetic workloads across diverse scenarios, all while providing accurate and reliable information reflecting the network’s reliability and integrity for each scenario.

*This document was prepared with the Navigation feature of Word in mind. It is recommended to open the Navigation pane when using this document as a reference.*

# Requisite Properties of ctsTraffic

There are 5 core requisite properties that ctsTraffic was designed from the beginning to satisfy. These qualities were derived from learning from prior internal and external tools and spending more time understanding the diversity of needs that Networking was being asked to satisfy.

* **Reliability** – our foremost requirement: we absolutely must have high confidence in the reliability and integrity of the tool and the information it is reporting; this level of reliability and integrity must scale to all scenarios we need to test.
* **Transparency**: we require tooling that provides a high degree of transparency into how the network is properly servicing the application. Notably, we require accurate details on connection integrity as well as data integrity – across any and all API usage patterns, any and all protocol patterns, and lastly across any degree of scale we need to push the tool.
* **Scalability**: we require tooling that can scale across a wide variety of targeted scenarios: we must be able to scale down (to very small devices and phones); we must be able to scale up (to greater numbers of CPUs; to greater available bandwidth); we must be able to scale out (to high numbers of concurrent connections). All scale targets must be met without sacrificing reliability or transparency.
* **API extensibility**: we need the ability to add new APIs and API calling patterns quickly and with low-risk without needing to build yet another highly reliable highly scalable Winsock app. This extensibility must not degrade the reliability and transparency requirements.
* **Protocol extensibility**: we need tooling that can be quickly and easily updated to reproduce various protocol patterns (patterns of sending & receiving data). This extensibility must meet all scalability targets, reliability targets, and scalability targets.

# Client & Server Options

ctsTraffic is designed as a classic client/server model, where the server will wait listening for client requests, while the client initiates connections to servers. ctsTraffic servers will accept any number of connections from any number of clients; ctsTraffic clients can target and make connections to one or more servers.

In order for the client and server to be in sync after making a connection, many of the same parameters must be specified on each respective command lines. This allows for a precise pattern of data to flow in expected directions between the client and server so each side can make the proper level of validation to assess the correctness of their scenario. This section covers the common and required options for client/server execution.

## Accessing Help

### -Help

Accessing help for ctsTraffic is targetable based on the intended scenario.

#### –Help

The default option displays information on the common [Server](#_Server-Specific_Options) and [Client](#_Client-Specific_Options) options.

#### –Help:tcp

This help option displays information detailing the various options specific to [TCP Scenarios](#_TCP_Scenarios).

#### –Help:udp

This help option displays information detailing the various options specific to [UDP Scenarios](#_UDP_Scenarios).

#### –Help:logging

This help option displays information detailing the various option one has in controlling the [Output Options](#_Output_Options): the formatting of the output, filenames for the desired output, and the verbosity of the output to console.

#### –Help:advanced

This options displays atypical scenario-specific [Advanced Options](#_Advanced_Options). These options were developed to support unique scenarios and are not expected to be used for typical usage patterns.

## Server-Specific Options

### -Listen

ctsTraffic functioning as a ‘server’ is defined by listening for incoming connection requests. The **–Listen** parameter can accept an IPv4 or IPv6 address (must be already assigned local to that machine), or it can accept **–Listen:\*** to listen on *any* address that is local to the system. Additionally, **–Listen** can be specified multiple times if there are multiple exclusive IP addresses that one wants to listen on. Yet another option with **–Listen** is one could specify to only listen on all IPv4 *or* all IPv6 addresses. For example, **–Listen:0.0.0.0** would listen for connections on all local IPv4 addresses.

The most common option is to specify **–Listen:\***, as most servers are not concerned with controlling which IP address a client can connect. Specifying individual IP addresses via one or more **–Listen** arguments is more common when one needs firm control over which interfaces are used for a scenario (*e.g. in a multi-homed scenario where only a subset of available interfaces are to be used*).

### -ServerExitLimit

This option exists to simplify automated deployments where it is fore-known exactly the number of connections an instance of ctsTraffic should handle before exiting.

#### Default value: -ServerExitLimit:0xffffffffffffffff (MAXULONGLONG – effectively infinite)

## Client-Specific Options

### -Connections

This option controls the total number of concurrent connections being made by client to the server(s) specified by **–Target**. For example, **–Connections:100** would instruct ctsTraffic will spin up a total of 100 connections to the specified server **–Target** addresses; once any connection completes, ctsTraffic will immediately create a new connection to maintain a total of 100 concurrent connections. This will repeat until ctsTraffic has completed a total number of connections equal to **–Connections** \* **–Iterations**.

#### Default value: -Connections:8

### -Iterations

This option controls the number of times ctsTraffic will cycle across all the connections specified with **–Connections**. Note that **–Iterations** is not itself the total aggregate number of connections a client will make to the **–Target** servers: the total number of connections is equal to **–Connections** \* **–Iterations**.

As an example, if **–Connections:16** **–Iterations:100** were specified, ctsTraffic would work to keep exactly 16 active connections established, making new connections once prior connections had completed, until a total of 1600 connections have been made, at which time the client would exit.

#### Default value: -Iterations:0xffffffffffffffff (MAXULONGLONG – effectively infinite)

### -Target

This option instructs the client which server(s) it should attempt to establish connections. This option allows for different methods to express the target server: one can specify the flat name of the server; one can specify the fully-qualified name of the server; one can specify an IPv4 address of the server; one can specify an IPv6 address of the server. Also note that multiple **–Target** command line arguments can be given to support targeting multiple servers or multiple specific IP addresses on a single server. Additionally, **–Target:loopback** will resolve to the known loopback IPv4 address of 127.0.0.1 and the know loopback IPv6 address of [::1] for when both client and server on the same machine.

When the client looks to make a new connection, it will iteratively target each IP addresses resolved from the one or more **–Target** options specified.

## Always-Required Options

While most parameters can be set individually on the client or server independently, there are a few parameters which must be in sync between the client and server. The following parameters *must be identical* on connected ctsTraffic client and server instances:

### -Port

The UDP or TCP port on which the server is listening must be established between the client and the server so the client will be able to successfully connect to the target server. When specified on the server command line, **–Port** will control the *local* port which the server will listen; when specified on the client command line, **–Port** will control the *target* port which the client will connect.

#### Default value: -Port:4444

### -Protocol

Both the client and server must agree to connect over TCP or UDP. The TCP protocol supports various protocol patterns via **–Pattern** controlling the flow of data being sent and received. The UDP protocol currently only supports a single protocol pattern: streaming from the server to the client.

#### Default value: -Protocol:TCP

### -Verify

Both the client and server must agree to perform the same level of validation of the connections established and data transferred. Verify currently supports two options:

#### -Verify:connection

The **–Verify:connection** level of verification instructs ctsTraffic *not* to look at the data being sent and received, only to look at the integrity of the connection and the expected number of bytes being successfully sent and received.

For TCP scenarios, not only will errors will be raised if a connection fails, but **–Verify:connection** will also raise errors if a connection receives more bytes than expected or if a connection terminates before transferring all the expected data.

For UDP scenarios, errors will be raised if Winsock functions fail. **–Verify:connection** will also log frames as dropped if the number of bytes in any frame isn’t the exact number of expected bytes as well as logging error frames when datagrams are received and stamped with an unexpected sequence number.

#### -Verify:data

The **–Verify:data** level of verification is a superset of**–Verify:connection**. In addition to the connection integrity checks, data validation will inspect and validate every received buffer of data against a known bit-pattern (the sender-side will always send from the same buffer pattern). Data is validated immediately upon successfully every receive IO request. Should any discrepancy be found, an error is immediately logged with such details and the connection is failed and terminated.

#### Default value: -Verify:data

## TCP-Required Parameters

An overview of these options is presented here as these are required on both client and server. See [TCP Scenarios](#_TCP_Scenarios) for a more thorough coverage of TCP-related options.

### -Pattern

For TCP connections, the wire-protocol must be agreed upon between the client and server so both sides of the connection remain in sync with regards to who should be sending how much data.

#### Default value: -Pattern:Push

### -Transfer

For TCP connections, the total number of bytes transferred must be pre-established between the client and the server in order to properly validate the connection integrity and data integrity of the scenario. Transfer represents the total number of bytes both sent and received, the direction of the data controlled by **–Pattern**.

#### Default value: -Transfer:0x40000000 (1 GByte)

## UDP-Required Parameters

An overview of these options is presented here as these are required on both client and server. See [UDP Scenarios](#_UDP_Scenarios) for a more thorough coverage of UDP-related options.

### -BitsPerSecond

For UDP connections, the bit-rate at which datagrams will be sent from the server to the client must be pre-established as the simple codec being used deliberately will process frames of data at a fixed rate (not adaptive to latency or drop rates).

### -FrameRate

For UDP connections, the frame rate by which datagrams are sent and processed must be pre-established to keep the client and server in sync throughout the UDP stream.

### -StreamLength

For UDP connections, the total length of the stream (in seconds) must be pre-established between the client and server to enable proper validation of all data sent and received.

**–BitsPerSecond** and **–FrameRate** work together to precisely control the flow of datagrams. For example, if **–BitsPerSecond:8000000 –FrameRate:100** were specified, 10000-byte datagrams will be sent 100 times per second (8000000 / 8 == 1000000 bytes per second, evenly split every 10 milliseconds –> 1 second / 100).

# TCP Scenarios

## API Usage Options

### -IO

ctsTraffic exposes 2 core API usage patterns that facilitate scaling across all scalability targets. The 2 Windows programming models are a) OVERLAPPED I/O using the Windows native Threadpool + IO Completion Ports, and b) the new queuing completion model of the Winsock Registered IO APIs.

#### -IO:IOCP

This is the default API usage pattern in ctsTraffic. It is the generally recommended API model to developers to scale well across a wide range of targets. This API model leverages OVERLAPPED I/O support from the NT operating system to manage individual IO requests and efficiently queue the completions to IO completion ports. ctsTraffic additionally employs the NT threadpool to distribute these IO completions across a pool of worker threads balanced across system resources, controlled and managed by the NT threadpool. Whereas developers were forced to manage and tweak their own threads in prior OS releases, it is recommended for modern development to leverage these new Windows threadpool APIs for a high performance, highly scalable thread pool implementation.

Consequently, ctsTraffic does not expose knobs to tweak or control how one uses “threads” with regards to IO requests: instead, IO is automatically efficiently distributed through the native optimizations implemented by the OS with IO completion ports and the NT threadpool.

This API pattern additionally supports handing successful completion in-line. The default behavior when using WSASend and WSARecv with OVERLAPPED I/O is for the IO completion handler to *always be notified*, even if the API call completes immediately (returns 0 – not WSA\_IO\_PENDING which is returned if the call cannot be immediately satisfied). This default behavior can incur additional overhead which can be particularly noticeable at various scale targets (e.g. extra CPU on really small devices; extra CPU when running a very high speeds). ctsTraffic utilizes an option first made available in Windows Vista: calling SetFileCompletionNotification with the flag FILE\_SKIP\_COMPLETION\_PORT\_ON\_SUCCESS. This flag will override the default Winsock behavior and when an OVERLAPPED API call completes immediately, the IO completion handler *will not be notified*.

***Note that this new flag can cause issue if there are Winsock Layered Service Providers (LSPs) installed. It is recommended not to run ctsTraffic with any LSPs installed: these 3rd party extension libraries utilize deprecated APIs and software shipping with these libraries are no longer given a Windows Logo.***

#### -IO:RIOIOCP

Windows 8 added new extensions to the Winsock 2.2 APIs dubbed ‘Registered IO’ APIs, or ‘RIO’. These APIs enable unique memory and IO models as compared to the existing Winsock APIs. Key differences include:

* All memory buffers in the app must first be “registered” using RIO APIs before being used to send or receive data. Registering a buffer will enable Winsock to ‘pin’ the buffers into physical memory ahead of any API call, thus avoiding the memory probe and lock costs associated with every buffer passed through the existing Winsock APIs. Since the memory buffers are pre-pinned, send and receive operations avoid these latency costs associated with the kernel memory manager.
* All IO requests move through a queuing-model versus the existing IO models exposed through Winsock APIs. Queues allow for efficient signaling between the application and the networking stack when IO requests are ready and when IO requests have been completed. Since standard eventing and IO manager infrastructure is no longer necessary, send and receive operations avoid these associated latency costs.

It should be noted that this new Winsock IO model with RIO APIs comes with tradeoffs that developers weigh as they architect and design their solutions. There are non-trivial memory costs associated with this model (since all buffers must be pre-pinned to physical memory) which can affect scale as one handles larger numbers of connections. Additionally, developers must also design and implement their own thread pool solution to scale across system resources. This includes designing and implementing proper synchronization around the RIO request queues and completion queues as well as designing how their threads are to be notified of IO completions. Thus, while RIO affords developers much lower latencies and predictable jitter rates, it also demands more low-level architectural design. For example, a developer must effectively balance the numbers of completion queues with the numbers of threads in their thread pools, as the application must provide locks around accessing these queues.

The RIO APIs were specifically targeting a subset of Enterprise scenarios where the workloads make extremely high numbers of send and receive requests with small buffers. These workload requirements demand a very low latency between one app sending and the other app receiving; most of the workloads additionally require a stable latency pattern with their IO requests (also called jitter).

It’s highly suggested therefore to keep the default option unless Registered IO is requisite for the customer deployment (*e.g. using RIO APIs as a high-performance IPC mechanism between services*).

##### Default value: -IO:IOCP'

## Protocol Options

As one of its requisite qualities, ctsTraffic has the ability to control many protocol characteristics network traffic and its protocols.

### -Buffer

An application’s use of the TCP protocol is impacted by how it supplies buffers to the networking stack to send or receive data. The size and frequency of buffers supplied by the application impact the characteristics of a connection as the TCP protocol works hard to manage optimal throughput via flow control and congestion control.

#### Default value: -Buffer:65536

The default buffer that is provided with every send or receive request is 64k bytes in size. This is quite reasonable for most workloads that don’t have other scenario requirements. Scenarios representing file transfers may want to use much larger buffers per request, while web scenarios may use smaller buffers per their usage models.

#### -Buffer:[1024,16384]

Additionally, ctsTraffic allows for a random-distribution of different buffer sizes used across its established TCP connections. This can be useful when there is a need to execute workloads simulating a diversity of IO requests. For example, one may want to choose from a set of small buffer sizes when simulating internet clients. This will instruct ctsTraffic to pick a random buffer size between these 2 values inclusively to be used for each new TCP connection.

### -Pattern

ctsTraffic exposes a variety of protocol patterns to leverage when establishing a scenario to execute. These patterns describe the flow of information between the client and the server across each individual TCP connection. Note every pattern has an interaction with the other ctsTraffic TCP parameters.

* The total number of bytes sent + received will equal the value specified with **–Transfer**.
* The buffer size used for each request will be referenced from the **–Buffer** parameter. Note that this can be over-ridden by the **–Pattern** if the pattern requires fewer bytes sent or received in the next IO request.
* When data validation is requested (**–Verify:Data**), buffers will be allocated for every connection to receive data (to enable data validation). When data validation is not requested (**–Verify:Connection**), a single static buffer is used for every connection, eliminating the per-connection memory overhead and optimizing for memory locality.
* Send requests will always pull from the same pre-allocated buffer which was pre-populated with a unique bit pattern. This avoids memory any overhead for per-connection buffer allocations while still allowing deep buffer validation when receiving this sent data.
* All send and receive requests indicated by the specified **–Pattern** will be executed through the API patterns controlled with the **–IO** option.

Also note that the **–Pattern** option must be identical on both the client and the server.

#### -Pattern:Push

This is the default TCP pattern. This pattern is similar to a “client upload” network usage pattern. Upon the client connecting to the server, the client will immediately begin sending data to the server. The server will continuously post receives until all expected data is received and validated.

#### -Pattern:Pull

This pattern is similar to a “client download” network usage pattern. Upon the client connecting to the server, the server will immediately begin sending data to the. The client will continuously post receives until all expected data is received and validated.

#### -Pattern:PushPull

This pattern is similar to ‘request-response’ protocols such as HTTP. Upon the client connecting to the server, the client and server will take turns sending and receiving data; at any point in time, data will only be traveling in one direction between the client and server, never being sent and received concurrently.

This alternating pattern starts with the client first sending data to the server (the number of bytes controlled by **–PushBytes**). After all bytes are successfully sent and received, the flow direction alternates with the server sending data to the client (the number of bytes controlled by **–PullBytes**). This pattern will continue to alternate between which side is sending or receiving until the total number of bytes (controlled with **–Transfer**) has completed.

Note that **–PushBytes** and **–PullBytes** interact with **–Buffer**: the smaller is used when controlling the size of each IO request.

##### -PushBytes

This parameter is only applicable when **–Pattern:PushPull** is specified. This parameter controls the number of bytes sent *from the client to the server* before the protocol alternates direction and the server begins sending to the client.

###### Default value: -PushBytes:1048576 (1MB)

##### -PullBytes

This parameter is only applicable when **–Pattern:PushPull** is specified. This parameter controls the number of bytes sent *from the server to the client* before the protocol alternates direction and the client begins sending data to the server.

###### Default value: -PushBytes:1048576 (1MB)

#### -Pattern:Duplex

This pattern exercises the full duplex sending and receiving of traffic concurrently between the client and server. Upon the client connecting to the server, both the client and server is simultaneously begin sending exactly ½ of the total bytes to transfer (controlled with **–Transfer**). The client and server will also be continuously posting receives to receive and validate data as it sends.

#### Default value: -Pattern:Push

### -RateLimit

This option is provided to enable users to carefully control the number of bytes being pushed across the wire when they don’t want to be running at full line-rate. **–RateLimit** is specified in terms of bytes/second, with the default being no rate limiting.

**–RateLimit** is implemented by splitting the total bytes/second into a sub-second slices and sending each slice at the precise sub-second time offset. The default is to split into 100 millisecond time slices. It does this to more evenly distribute the rate-limited traffic across each 1-second period.

For example, if one wanted to Rate-limit a connection to resemble US ISP cable connectivity, one might specify **–RateLimit:2097152** (2 MB/sec, or 16 Mbits/sec). As the default rate distribution is split across 100ms time slices, ctsTraffic will send 209715 bytes every 100ms.

#### Default Value: -RateLimit:0 (no rate limits)

### -Transfer

This option controls the total number of bytes being sent and received across every TCP connection. The default value is 1 GB (1073741824 bytes). This total is factored into the **–Pattern** specified.

Note that this option must be identical on both the client and the server.

#### Default value: -Transfer:0x40000000 (1 GByte)

## Status Updates

The status output for TCP connections collect data points across 6 areas: Send bytes/sec, Receive bytes/sec, In-flight connections, Completed connections, Network Errors, and Data Errors.

An example output using the Duplex TCP protocol pattern over loopback looks like this (removing the connection data). This output is from ctsTraffic running as a server with status output set to write every 1000 milliseconds.

ctsTraffic.exe –Listen:\* -Pattern:Duplex –StatusUpdate:1000 –ConsoleVerbosity:1 –StatusUpdate:1000

Legend:

\* TimeSlice - (seconds) cumulative runtime

\* Send & Recv Rates - bytes/sec that were transferred within the TimeSlice period

\* In-Flight - count of established connections transmitting IO pattern data

\* Completed - cumulative count of successfully completed IO patterns

\* Network Errors - cumulative count of failed IO patterns due to Winsock errors

\* Data Errors - cumulative count of failed IO patterns due to data errors

TimeSlice SendBps RecvBps In-Flight Completed NetError DataError

1.0 234449966 178277166 8 0 0 0

2.0 232128512 180549440 8 0 0 0

3.0 208327224 189645394 8 0 0 0

4.0 183159671 187108484 8 0 0 0

5.0 185860096 196790056 8 0 0 0

6.0 200929054 157865278 8 0 0 0

7.0 182124544 191705460 8 0 0 0

8.0 183697408 163743876 8 0 0 0

9.0 199163904 186973896 8 0 0 0

10.0 175308800 193139876 8 0 0 0

11.0 170655744 152012496 8 0 0 0

12.0 192086016 190928320 8 0 0 0

13.0 148111360 166126132 8 0 0 0

14.0 176881664 165917844 8 0 0 0

15.0 198705152 187278856 8 0 0 0

16.0 174063616 193212776 8 0 0 0

17.0 192282624 171599528 8 0 0 0

18.0 183697408 189525108 8 0 0 0

19.0 167182336 188512032 8 0 0 0

20.0 204013568 163306444 8 0 0 0

21.0 159318016 213575492 7 1 0 0

22.0 193724416 161491088 7 1 0 0

23.0 148504576 172160152 5 3 0 0

23.6 204161 238603457 0 8 0 0

### Timeslice

This column identifies the time-offset from the start of the test. This allows for easier understanding of time slice deltas and the processing of information over a specified amount of time.

### SendBps

This column calculates the number of bytes/second that were *sent* within the specified time slice. Here, the total # of bytes transferred are divided by 1 second before printing (since **–StatusUpdate** was 1 second).

### RecvBps

This column calculates the number of bytes/second that were *received* within the specified time slice. Here, the total # of bytes transferred are divided by 1 second before printing (since **–StatusUpdate** was 1 second).

### In-Flight

This column reflects the *current number* of TCP connections transferring data. In the above example, you can see that at 17 seconds, 2 TCP connections completed, dropping the In-Flight column down by two to 8 active TCP connections.

### Completed

This column reflects the *aggregate of all successful* TCP connections since the start of the test. As opposed to the first 3 columns, this column does not reflect solely changes that occurred within the prior time slice, but maintains a total count.

### NetError

This column reflects the *aggregate of all previously failed* TCP connections since the start of the test – specifically when failed due to a network error (*e.g. a failure from a Winsock API that thus caused the connection to be terminated*). As opposed to the first 3 columns, this column does not reflect solely changes that occurred within the prior time slice, but maintains a total count.

### DataError

This column reflects the *aggregate of all previously failed* TCP connections since the start of the test – specifically when failed due to a data validation error (*e.g. connections which transmit too few or too many bytes before terminating; connections which are found to have received corrupt data*). As opposed to the first 3 columns, this column does not reflect solely changes that occurred within the prior time slice, but maintains a total count.

# UDP Scenarios

## API Usage Patterns

Currently ctsTraffic supports one API usage model with UDP datagrams: one which follows the semantics and usage found with media streaming (audio streams and video streams).

The server-side of the UDP ‘connection’ uses a timer wheel based off the NT threadpool APIs to guarantee datagrams are sent at specific offsets in time. Upon the timer firing, the NT threadpool calls ctsTraffic on one of its threadpool thread, where ctsTraffic makes a blocking send call. These calls are blocking as Winsock does an effective job buffering and as these are set on a specific timer, there is not a need to unblock these threads and incur the latency for the send completions.

The client-side of the UDP ‘connection’ uses IO completion ports to receive data, uses the NT threadpool to distribute these receive calls on available threads, and handles inline completions to avoid unnecessary delays in processing data. In-line completions is an option that was added in Windows to allow APIs that use OVERLAPPED I/O (such as WSARecvFrom used in ctsTraffic) to let the IO request pend asynchronously if data is not ready to complete the request (which would complete in the IO completion port) *or* if the data is immediately ready to satisfy the IO request at the time it’s made, complete the IO request in-line on an efficient fast-path.

## Protocol Patterns

ctsTraffic exposes a single UDP protocol pattern to facilitate crafting scenarios where the server sending datagrams to a client at a fixed size and rate. This is done to synthesize UDP protocol usage with audio and video streaming codecs while being able to measure for fixed target goals.

Whereas many modern media codecs will leverage a variable bitrate to dynamically adjust the sending patterns based on real-time network characteristics of throughput, latency, and loss, ctsTraffic will instead guarantee a precise and predictable streaming behavior throughout the lifetime of the stream based on the user’s options for validating the ability to sustain the specified streaming characteristics.

In order to provide deeper validation, every datagram of data that is sent from the server includes a frame number. This allows the client to appropriately track every datagram received into a ring-buffer where it processes those frames at the specified frame rate.

As it is a common semantic for UDP streaming protocols to describe their network characteristics in terms of bits/second and frame rate, ctsTraffic surfaces that on the command line as options for this protocol.

### -BitsPerSecond

This argument is required on both the server and the client, and it must match for proper validation of the tested UDP stream. ctsTraffic uses **–BitsPerSecond** to guarantee the correct buffered bytes are sent by the server every per-second interval, while the client uses the value to verify the expected size of every frame from the datagrams received from the server.

### -FrameRate

This argument is required on both the server and the client, and it must match for proper validation of the tested UDP stream. **–FrameRate** is defined as ‘the number of frames per second’: the frequency that one or more datagrams will arrive to satisfy one frame of the stream. Thus in terms of network behavior:

* one frame is sent every (1000 / **–FrameRate**) milliseconds
* one frame == (**–BitsPerSecond** / 8 / **–FrameRate**) bytes

For example, if a user were to specify **–BitsPerSecond:8000000 –FrameRate:50**, a 20,000-byte datagram would be sent from the server 50 times/second (every 20 ms).

It should also be considered that one datagram can be at most 64kB in size – which means that frames that are larger than 64kB will be a burst of consecutive datagrams sent to satisfy that frame requirement. The client will verify that a frame is received only when all required bytes for that frame have been received.

### -StreamLength

This command line argument is required on both the server and the client, and it must match for proper validation of the tested UDP stream. **–StreamLength** specifies in the number of *seconds* which a stream will be sent from the server, and which the client will be processing expected frames.

### -BufferDepth

This command line argument is only required on the client. **–BufferDepth** indicates to the client in terms of *seconds* the amount of initial time to allow for ‘buffering’ of datagrams before initiating its timer wheel to start processing frames. This option allows for some degree of variance in the network as frames are being transferred, and has common usage with real-world streaming protocols for this very purpose.

### -StreamCodec

This command line argument optional only for the client. The ctsTraffic UDP protocol supports 2 basic ‘codecs’.

#### –StreamCodec:NoResends

This codec makes no effort to recover any dropped datagrams. When this codec is specified, all missing frames are marked as dropped as the client’s timer wheel processes frames (based off of **–FrameRate**).

#### –StreamCodec:ResendOnce

This codec will make a single best-effort attempt to re-request a dropped frame. When this codec is specified, the client will ‘peek’ ahead into its ring buffer (the size of that ring buffer is controlled by –**BufferDepth**) to check if that buffered frame was indeed received. If that frame has not been received, the client will immediately send an out-of-band request back to the server to resend that missing frame.

#### Default value: –StreamCodec:NoResends

## Status Updates

As an example, to test an HD video stream scenario which transmits at 10Mbits/sec & 24 frames/sec, the ctsTraffic command line would contain:

ctsTraffic.exe –Listen:\* –Protocol:Udp –BitsPerSecond:10000000 –FrameRate:24 –BufferDepth:4 –StreamLength:3600 –StatusUpdate:1000

*Note: for this example, the ctsTraffic client would have an identical command line, replacing –Listen with –Target.*

*Note: due to odd font issues, be careful not to copy and paste directly from Word into a command shell window – unexpected / unseen characters will also be pasted in.*

The ctsTraffic server will send one frame of data (52083 bytes) 24 times per second. As this number of bytes will fit in one datagram, this would result in one 52,083 byte datagram sent every ~41 milliseconds. The server will repeat this pattern of sending 52,083 24 times/second for 3600 seconds (1 hour).

The ctsTraffic client would receive these same parameters, and would have a timer-wheel fire at that same cadence (24 times/second) expecting to have received the next full frame of data at each tick interval.

Status output for the scenario tracks both bits/second and how frame are processed.

TimeSlice Bits/Sec Completed Dropped Repeated Retries Errors

1.0 9516655 0 0 0 0 0

2.0 9999936 0 0 0 0 0

3.0 9999936 0 0 0 0 0

4.0 9989946 2 0 0 0 0

5.0 9999936 24 0 0 0 0

6.0 9999936 24 0 0 0 0

7.0 9999936 24 0 0 0 0

8.0 9999936 24 0 0 0 0

9.0 9999936 24 0 0 0 0

10.0 9999936 24 0 0 0 0

11.0 9999936 24 0 0 0 0

12.0 9999936 24 0 0 0 0

13.0 9989946 24 0 0 0 0

14.0 9999936 24 0 0 0 0

15.0 9999936 24 0 0 0 0

16.0 9999936 24 0 0 0 0

17.0 9999936 24 0 0 0 0

### TimeSlice

This column identifies the time-offset from the start of the test. This allows for easier understanding of time slice deltas and the processing of information over a specified amount of time.

### Bits/Sec

This column captures the number of bits/second (bytes \* 8) that were *received* within the specified time slice. Here, the total # of bits transferred are divided by 1 second before printing (since StatusUpdate was 1 second).

### Completed

This column captures the number of frames successfully processed within that time slice. As FrameRate was set to 24, it’s expected that 24 will be in the Completed column, as StatusFrequency was set to 1 second.

### Dropped

This column captures the number of frames that were dropped within that time slice. Dropped is determined by the client’s timer wheel firing to ‘render’ the next frame, but our ring-buffer shows that the datagram for that frame had not been received.

### Repeated

This column captures the number of frames which were ‘repeated’ – meaning any instances where the datagrams with the same frame number were received on the client. This generally indicates an issue with the networking stack or a complex routing topology where instances of datagrams being duplicated could occur.

### Retries

This column captures the number of frames where the “ResendOnce” StreamCodec peeked ahead into the ring-buffer and found a frame that had not yet arrived (though was expected to be there). With –StreamCodec:ResendOnce specified, a request to the server would have immediately been sent request that missing frame be resent.

If this resend process was successful (the frame did indeed arrive before the timer wheel processed it), it is counted in the ‘Retries’ column. If the resend process did not succeed (the frame was still not present when the timer wheel processed it even though it re-requested it), it will be counted in the ‘Dropped’ column.

### Errors

This column captures any errors that have occurred on streams that were running during that time slice. Errors for datagrams may include network errors (*e.g. Winsock APIs like WSARecvFrom failing*) or data errors (*e.g. data corruption in a datagram was discovered*).

# Output Options

ctsTraffic can provide detailed information about the session in a variety formats with different data points depending on desired information. The information available falls into 4 categories: connection details, error details, status updates, and jitter data.

When wanting to write to the console, these options can be controlled with **–ConsoleVerbosity**. This option controls what combination of the available output options will be written.

## -ConsoleVerbosity

There are 6 options one can specify to control the verbosity of information written to the console.

### -ConsoleVerbosity:0

With option 0, *nothing* is written to the console.

### -ConsoleVerbosity:1

With option 1, only status updates are written to the console.

### -ConsoleVerbosity:2

With option 2, only error details are written to the console.

### -ConsoleVerbosity:3

With option 3, only connection information is written to the console.

### -ConsoleVerbosity:4

With option 4, both connection information and error information are written to the console.

### -ConsoleVerbosity:5

With option 5, connection information, error details, and status updates are all written to the console.

### Default value: -ConsoleOutput:3

Additional options exist when wanting to directly write this information to file: **–ConnectionFilename**, **–ErrorFilename**, **–StatusFilename**, and **–JitterFilename**. Any set of these options can be specified, with the default being *no* files are created. The filenames for those options specified can be unique per option (generally recommended), or one could specific the same filename for all 4 options if one wanted to have all data points interleaved into the same file (though that’s not recommended). Splitting information across files can assist in post-processing details of what all occurred in that ctsTraffic session.

Additionally, the file extension of filenames specified for the parameters **–ConnectionFilename** and **–StatusFilename** will control the formatting of the output written.

* A .csv file extension will have all status updates written in a comma-separated-values format, including column headers.
* A .wtl file extension will direct all output to a WTT Logfile. Note that this option will require the following Windows test binaries: wttlog.dll, Wex.Logger.dll, Wex.Common.dll, and Wex.Communication.dll.
  + **Note: this option is not available in the public release as these libraries are internal to Microsoft.**
* Any other file extension will be written as clear text.

**–ErrorFilename** will always be written as clear text and **–JitterFilename** will always be written as its own specified csv format.

A time-stamp is written for all connection, error, and status data points (jitter doesn’t need this as it has its own format-specific log). The time-stamp is recorded as the number of seconds relative to the start of the application and is consistent across all log files and console. This helps in correlating events across files.

## **Connection information**

Connection details are written at connection establishment and connection closure. The connection details include the source IP and port, the destination IP and port, the success or failure of the connection, and the error information for any errors that occurred. The closure details will include additional information specific to the protocol that was tested.

This is sample output with TCP default options (ctsTraffic.exe –Listen:\*) after successfully completing a single connection from a client:

[0.0] TCP connection established to [::1]:4444

[1.0] TCP connection completed successfully : [[::1]:5674 - [::1]:4444] : SendBytes[1073741824] SendBps[1030462403] RecvBytes[0] RecvBps[0] Time[1042 ms]

You’ll notice both data points begin with the time-offset in seconds within a square bracket. Information written after the address tuples is protocol-specific (specific to TCP in this case), providing the summary of that one connection. Some of these same data points tracked per connection are displayed with Status Updates.

## Error details

Information can be logged at the point when an error occurs, specifying the relevant details (the API that failed with the error code, for example). This can be useful when troubleshooting failures to better understand the context at the point when something fails.

This is sample output with TCP default options for the duplex pattern (ctsTraffic.exe –Listen:\* -Pattern:duplex) when the client is aborted early. You’ll notice the connection details are interleaved with the error details; this is the default console logging verbosity. When multiple errors occur (in this case, both the WSASend and WSARecv failed when the client RST the connection), the first error is captured in the Connection details. This design decision was made to ensure the first error was not masked by a potential waterfall of other errors which might hide the original point of failure.

[8.0] TCP connection established to [::1]:5721

[13.8] WSASend failed (10054) An existing connection was forcibly closed by the remote host.

[13.8] WSARecv failed (10053) An established connection was aborted by the software in your host machine.

[13.8] TCP connection failed with the error code 10054 : [[::1]:4444 - [::1]:5721] : SendBytes[306839552] SendBps[52857803] RecvBytes[247398400] RecvBps[42618156] Time[5805 ms]

Errors will generally fall into 2 categories: an API failure while executing a protocol pattern (such as the above failures in WSASend and WSARecv); a validation failure while processing sent and received data. The former error type will present the error information as reported from the underlying APIs that failed; the latter is specific to ctsTraffic and will have additional information specific to the logic failure that was detected (these failure types are listed below).

[5.4] Connection aborted due to the protocol error ErrorNotAllDataTransferred

[5.4] WSARecv failed (10054) An existing connection was forcibly closed by the remote host.

Depending on the level of verification occurring (**–Verify:Connection** or **–Verify:Data**), one might see one of the below validation logic errors written should any of the connection or data integrity checks fail.

#### ErrorNotAllDataTransferred

This error can be returned with either **–Verify:Connection** or **–Verify:Data** options.

This error occurs when a TCP connection is cleanly shutdown (not aborted/RST) but that end of the connection did not yet send or receive all the data expected as controlled by the **–Transfer** command line option.

This error can occur if the client and server do not match their **–Transfer** command-line arguments, causing one side to expect to transfer fewer bytes than the other.

#### ErrorTooMuchDataTransferred

This error can be returned with either **–Verify:Connection** or **–Verify:Data** options.

This error occurs when a TCP connection is cleanly shutdown (not aborted/RST) but that end of the connection received more data than expected as controlled by the **–Transfer** command line option.

This error can occur if the client and server do not match their **–Transfer** command-line arguments, causing one side to expect to transfer more bytes than the other.

#### ErrorDataDidNotMatchBitPattern

This error will only be raised when **–Verify:Data** is specified.

This error occurs when data received does not match the ctsTraffic bit-pattern that is expected to be sent.

This error can occur if the client and server do not agree on **–Verify** settings. As the known bit-pattern will not be tracked if **–Verify:Data** is not specified, the data may be incorrect for the side that did specify **–Verify:Data** (and thus is expecting bit-patterns to be correctly tracked).

A more serious scenario where this error can occur is when the data is genuinely being corrupt between the sender and the receiver. The error message will write out the buffer pointers of the expected data and corrupted data, as well as writing out the exact offset which corruption occurs. An example of this failure could look like this:

[1258.2] ctsIOPattern found data corruption: detected an invalid byte pattern in the returned buffer (length 59933): buffer received (000000A81683A5A0), expected buffer pattern (000000A817360091) - mismatch from expected pattern at offset (12432) [expected character '0x0' didn't match '0x68']

Because of this, if both client and server specify **–Verify:Data**, it’s recommended to also specify **–OnError:Break**, which will indicate to ctsTraffic to fail fast into the debugger when an error occurs. Once in the debugger, one can further debug the corruption by using the output information to dump out the buffers to observe the bad data and further troubleshoot the issue.

## Status Updates

Information can also be logged to convey the runtime state during execution. Updates are written to console and/or file at a specified period, with information being printed expressing changes in what has transpired over that previous period (for example, the number of bytes that were sent and received since the previous status update).

The details of the status update vary depending on the protocol being tested ([Status Updates](#_Status_Updates) under TCP Scenarios details TCP status update information; [Status Updates](#_Status_Updates_1) under UDP Scenarios details UDP status update information).

#### -StatusUpdate

This controls the frequency that the status updates are written, specified in milliseconds. It’s recommended to not set below 1000 milliseconds to avoid logging costs affecting the overall performance of the scenario.

#### Default value: -StatusUpdate:5000 (every 5 seconds)

# Advanced Options

Additional options are available that were developed to target unique scenario requirements. These options are not required for the most common scenarios and thus are listed separately.

## Unique API Options

As ctsTraffic was developed with API extensibility as a first-class requirement, additional API patterns are available for such scenarios with those unique requirements.

### -Acc

This parameter is made available to control the APIs used to accept incoming TCP connections to a server from a listening socket.

#### -Acc:AcceptEx

The default value is AcceptEx: pre-posting asynchronous (OVERLAPPED) accept requests. This API usage scales very well as it does not require dedicated threads to be created for each call and can service high rates of incoming requests for busy servers.

#### -Acc:accept

Winsock also exposes a blocking API call to accept TCP connections: accept(). This API usage requires a dedicated thread as it will block until a connection arrives to be serviced. As this blocking behavior can affect scalability, it is recommended to use this only in scenarios where one is not expecting to service large numbers of incoming connection requests.

### -Conn

This parameter is made available to control the APIs used to make outbound TCP connections to a server.

#### -Conn:ConnectEx

The default value is ConnectEx: pre-posting asynchronous (OVERLAPPED) connect requests. This API usage scales very well as it does not require dedicated threads to be created for each call and can scale to satisfy high numbers of concurrent connect requests.

#### -Conn:connect

Winsock also exposes a blocking API call to establish outbound TCP connections: connect(). This API usage requires a dedicated thread as it will block until the connection attempt completes successfully or fails. As this blocking behavior can affect scalability, it is recommended to use this only in scenarios where one is not expecting to make large numbers of outbound connections.

### -IO

This parameter is made available to offer API patterns conducting IO in addition to the above [TCP –IO](#_-IO) options *(iocp for using Winsock APIs with OVERLAPPED I/O leveraging IO completion ports; rioiocp for using the Winsock Registered I/O APIs with IO completion ports for notification)*.

#### -IO:ReadWriteFile

As Winsock SOCKET handles create FILE\_OBJECTs in the Winsock kernel, the Winsock file APIs ReadFile and WriteFile can be used directly on SOCKET handles. This IO option will exercise the scenario where ReadFile (used to receive from a socket) and WriteFile (used to send on a socket) are called asynchronously using OVERLAPPED I/O and IO completion port completion routines.

### -Options

By default, ctsTraffic does not set any specific socket option (via setsockopt()), change the IO mode of a socket (via ioctlsocket()), or make any IOCTL call on a socket (via WSAIoctl) with one exception: all server TCP sockets have keep-alives enabled.

#### -Option:keepalive

This option will enable keep-alive probes on all TCP sockets. Keep-alive probes are beneficial to detect when the remote TCP endpoint is no longer available – notably when that remote endpoint did not RST or FIN the connection before going away. This occurs in a variety of real-world scenarios. For example, if a router between the 2 endpoints fails and another route cannot be found, all packets are dropped. Another scenario is if the remote endpoint bugchecks (blue screens) or loses power.

These scenarios can be problematic when attempting to receive from that remote endpoint, as that recv will not be notified the remote endpoint is down and will just wait for data – as it never receives a RST or FIN. As all send requests are required to be acknowledged (ACK’d), send requests are not affected by this TCP behavior.

The keep-alive option will instruct the TCP/IP stack to re-request the previous ACK when it has not seen any packets from that remote endpoint for a period of time.

#### -Option:TcpFastPath

This option was added to Windows with the Windows 8 release. It is exclusive to TCP over loopback, where both side of the TCP connection each set this socket option. In this scenario, this option will allow the TCP/IP stack to take a highly optimized path, increasing throughput availability and lowering latencies.

## Unique Runtime Behavior Options

Additional options are provided for users who need specific custom network behavior provided with ctsTraffic.

### -OnError

This is provided to enable the user to control how ctsTraffic reacts when an error is encountered (‘an error’ being anything that would result in a failure on a socket and thus being written to the error logfile).

#### -OnError:Log

The default behavior is to write the error to the console (if **–ConsoleVerbosity** allows for error information) and/or write to the error log file (if **–ErrorFilename** was specified).

#### -OnError:Break

This option is added to facilitate troubleshooting by breaking into the debugger when an error occurs. This option can be useful when trying to capture the state when an error occurs or when debugging corruption.

Note that it is highly recommended to have ctsTraffic already running under a user-mode debugger (such as ntsd, cdb, or WinDbg) as this break will be fatal (it’s a 2nd-chance exception which will immediately terminate the process if a debugger is not there to debug the process). A kernel debugger can also be used if the deployment has it setup, though this is not required.

### -PrePostRecvs

By default, ctsTraffic will keep one outstanding receive call for all TCP and UDP connections. This generally scales well and is the most common coding pattern. This option allows for more than one receive to be outstanding on all sockets. For example, if **–PrePostRecvs:2** is specified, every connection will have its receive API called multiple times to ensure that there are 2 pending calls at all times. When used judiciously, this option can allow for scaling up to greater throughput. As with all things Performance related, there aren’t guarantees here ☺.

### -RateLimitPeriod

This option direct affects how **–RateLimit** behaves. **–RateLimit** throttles throughput across TCP connections to the specified number of bytes per second. It does this by slicing each 1-second period into smaller periods and throttles to those sub-second periods. For example, specifying **–RateLimit:100000** will ensure that no more than 100,000 bytes will be sent each second. It does so by throttling to each 1/100th of a second (by default) – working to keep each 1/100th of a second sending only up to 1000 bytes. It performs this way to keep throughput more predictable and avoid large peaks and valleys.

**–RateLimitPeriod** controls this per-second slicing value. As referenced in the above example, the default value is 100. Increasing **–RateLimitPeriod** will create an even more predictable flow over time (as the flow rate will be measured within smaller timeframes) and increasing this value will potentially allow for greater variance in the throughput of the connection.

Note that **–Buffer** will always take precedence over the suggested number of bytes to send as calculated with **–RateLimitPeriod** and **–RateLimitPeriod**. For example, if the user has specified **–Buffer:10000** but specified **–RateLimit:100000 –RateLimitPeriod:100** (which means each 1/100th of a second up to 1000 bytes will be sent). Since the buffer is 10,000 bytes, the start of the 1-second period will send the full 10,000 bytes (respecting **–Buffer** first), but will not send any data for the next 9 1/100th of a second periods (since 10,000 bytes will be sent, it will wait until the **–RateLimit** allows for more data to be sent).

### -ThrottleConnections

This option allows the user to change the gate behavior of ctsTraffic when making output TCP connections. By default, ctsTraffic will allow up to 1000 outstanding connection requests. It does this to have a more graceful build-up of connections to reach the target number of connections. If the scenario requires, this value can be lowered (for a longer build-up) or raised (to allow for a more resource demanding burst of connection requests).

### -TimeLimit

This options allows for exiting the application based off expired time as a “last-resort”. The default and suggested behavior is to control the lifetime of the instance of ctsTraffic based on the work it was asked to perform. For example, the client would control this via **–Connections** and **–Iterations**, while the server would control this via **–ServerExitLimit**. If the time limit is hit before the settings specified complete the scenario, the process will exit with errors logged when any existing connections were aborted.

## Unique Deployment Options

The above options were designed to satisfy them majority of deployment scenarios. These additional options are provided to enable scenarios that had unique requirements.

### -Bind

The default behavior for clients is to not bind to any one specific address before trying to connect to a target, but allow the stack to perform an *implicit bind*: meaning ctsTraffic will ask the stack to choose the most appropriate interface and address to use to connect to the target address (based primarily off the routing table). This option gives the end-user the control to specify exactly which address (or addresses) to exclusively use when making outbound connections.

This is useful in multi-homed scenarios where one wants to have immediate control over which interface is used to establish a connection.

One should be cautious using this option. ctsTraffic will round-robin through all addresses given to **–Bind** which match the target IP address class (IPv4 or IPv6) and will not query to see if there is a route from the chosen bind address to the target address. The user should ensure that routes exist from all addresses passed to **–Bind** and all addresses resolved to **–Target**.

### -Compartment

This option was added for a new feature added to Windows Server 2012 R2: the ability to add IP Interface compartments on Virtual NICs in a Hyper-V datacenter deployment. The targeted scenario for this option is for virtual gateway deployments, where the datacenter hoster wants to consolidate their gateway VMs bridging the hosted virtual networks with the outside world. Compartments maintain complete IP isolation between the hosted virtual networks, enabling a router, NAT, or Site-to-Site VPN gateway for multiple tenants, all on a single gateway Hyper-V deployment.

This option expected the Interface Alias (“ifAlias”) of the IP Interface Compartment desired to be targeted. This Interface alias will affect all Winsock and network usage, including IP addresses for binding, routing tables for connecting, and neighbors for direct communication.

The Interface Alias can be easily discovered through Powershell commandlets, such as “Get-NetIPConfiguration –detailed” or “Get-NetIPInterface”.

### -LocalPort

The default behavior for clients is to not bind to any one specific port before trying to connect to a target, but allow the stack to find an available port on its behalf. This is the highly suggested method clients should use as very few client application should rely on having exclusive access to a TCP or UDP port. There are scenarios which exist which *do* have such requirements. This option will attempt to bind all local sockets to this port before trying to connect. Use cautiously, as ports can only be used once. Therefore very few concurrent connections can be established by using the same local port.