APPENDIX C

QoS Measurement: qtcp

qtcp is a tool that can be used to measure end-to-end network service quality with a high degree of precision. qtcp can be used to examine the service provided by a network to the traffic of an existing application or, alternatively, can be used to examine the service provided to a user-defined traffic profile. qtcp works by timestamping packets at both sender and receiver, and computing packet transit delay variation.

Note that qtcp does not measure fixed components of transit delay (such as speed of light delay and serialization delay). Instead, qtcp measures the variable component of transit delay, which is primarily attributable to queuing in forwarding devices. QoS mechanisms affect only this variable component of transit delay. Therefore, from a QoS perspective, this is the more interesting component to measure.

Note

One of the strengths of qtcp is that it yields *per-packet* latency measurements rather than the kind of summary statistics offered by alternate network characterization tools. Although summary statistics are often useful (especially for handling very large data sets), they tend to obscure detailed network behavior. Detailed network behavior is important to understand when analyzing the performance of multimedia applications and the corresponding QoS mechanisms.

For optimal results, qtcp relies on a kernel timestamping module that can be run only on Windows 2000 (and later versions of the operating system). qtcp may be used either on a production network or on a controlled network that may be artificially congested.

qtcp offers the following features:

- qtcp is capable of reporting very precise delay variations, on the order of microseconds.
- qtcp invokes network QoS (by default, in test stream mode) and is useful for the purpose of evaluating QoS mechanisms.
- qtcp can measure delay variations experienced by existing application traffic.
- qtcp can simulate traffic flows for a range of user-selectable packet sizes and can measure the experienced delay variations.
- qtcp can simulate traffic flows shaped to a range of token bucket parameters and can measure the experienced delay variations.
- qtcp can be used on an isolated, controlled network or a production network.
- qtcp generates logs detailing packet-by-packet delays.
- qtcp collates results and generates various statistics.

This appendix describes the qtcp tool, explains how it works, and discusses interpretation of qtcp results. It follows with detailed usage guidelines and sample results.

Disclaimer

Microsoft periodically updates tools such as qtcp and associated components. As a result, there will be variations in functionality and usage from time to time. This appendix represents the state of qtcp at a particular snapshot in time and cannot be assured to represent any specific release of the tool. Up-to-date, release-specific information should be obtained directly from Microsoft.

Network Under Test

qtcp can be used in a production network or in a controlled network that may be artificially congested. Figure C.1 illustrates the usage of qtcp in a production network.

In this diagram, a qtcp sender and a qtcp receiver are located at opposite ends of the network. The network under test is a production network. As a result, other senders and receivers compete with the qtcp session for network resources. In this scenario, the qtcp user has no control over the current network load. Devices in the network under test may provide Quality of Service through any number of QoS mechanisms, or may not provide Quality of Service at all.

Alternatively, qtcp can be used in a controlled network, as illustrated in Figure C.2.

Figure C.1 qtcp in a Production Network

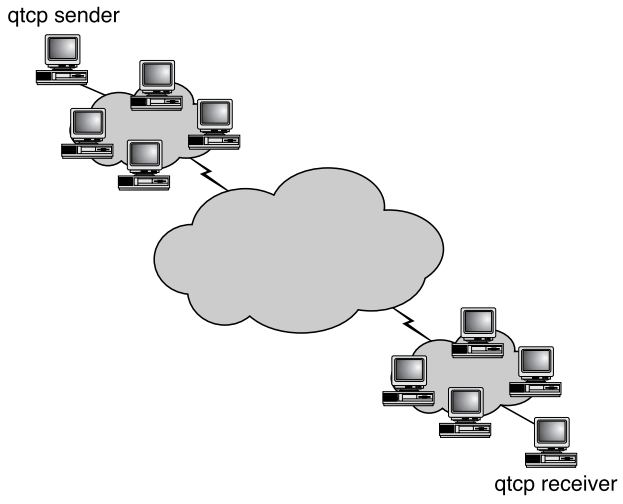
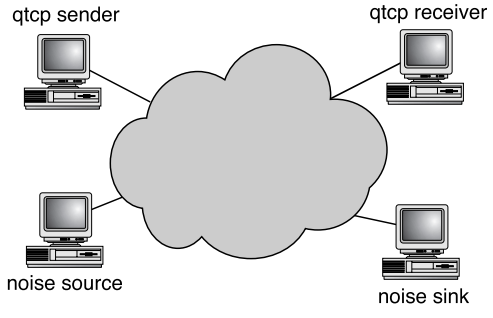


Figure C.2 qtcp in a Controlled Network

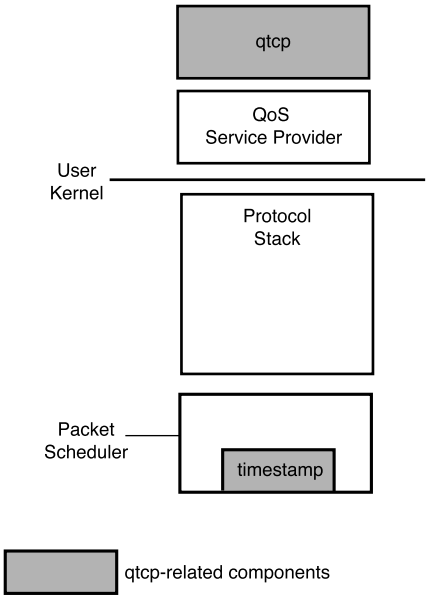


In this example, the qtcp sender and receiver are again located at the opposite ends of the network under test. However, the network under test is smaller and simpler, to facilitate the evaluation of specific network elements in isolation. Note that, unlike the production network, only four hosts are attached to this network. Two hosts are used to run the qtcp test session. The other two hosts act as a noise source and noise sink. In these type of tests, the user controls the amount of noise generated across the network during any specific test run. Generally, noise sources are selected to mimic the distribution of packet sizes and the traffic level that would be encountered on a production network. Noise sources need not be hosts. The noise source and sink illustrated could be replaced by a hardware noise generator.

How It Works

qtcp is implemented as a user-level application that runs on both a sending and a receiving host. For optimal results, it relies on a kernel resident timestamping module that is installed with the QoS packet scheduler on each host. These components are illustrated in Figure C.3.

Figure C.3 qtcp-Related Components



qtcp operates in two phases: the *runtime* phase and the *post-processing* phase. During the runtime phase, a traffic stream is sent from the qtcp sender to the qtcp receiver. (In addition, a TCP control channel is established between sender and receiver.) During this phase, qtcp collects a list of timestamps indicating the time at which each packet in the traffic stream is sent to the network and received from the network. In the post-processing phase, qtcp correlates sent and received timestamps, and calculates latencies and related statistics.

Operating Modes

qtcp operates in one of two modes: *test stream mode* and *application stream mode*. Application stream mode qtcp is also known as *Jittermon*. Test stream mode is used to investigate the impact of the network on a test traffic stream having a user-defined profile. Application stream mode is used to investigate the impact of the network on the traffic

stream generated by an existing application. The two modes differ in terms of the runtime behavior of qtcp. The post-processing behavior is the same, regardless of the operating mode.

Test Stream Mode

In test stream mode, the user invokes qtcp from a command line, specifying (among other parameters) the packet size to be used and the temporal profile of the test packet stream. qtcp generates buffers of the appropriate size for transmission to the kernel network stack. It also invokes traffic control to schedule packets in conformance with the specified profile.

QoS in Test Stream Mode

The default behavior of qtcp in test stream mode is to invoke full QoS (via the GQoS API), including RSVP signaling, traffic shaping, and packet marking. (See Chapter 12, "The GQoS API and the QoS Service Provider," for details on RSVP signaling, and see Chapter 13, "The Traffic Control API and Traffic Control Components," for details on traffic shaping and packet marking.) Command-line options enable the user to modify or bypass various components of QoS. See the section "How to Use qtcp," later in this chapter, for further detail.

Note

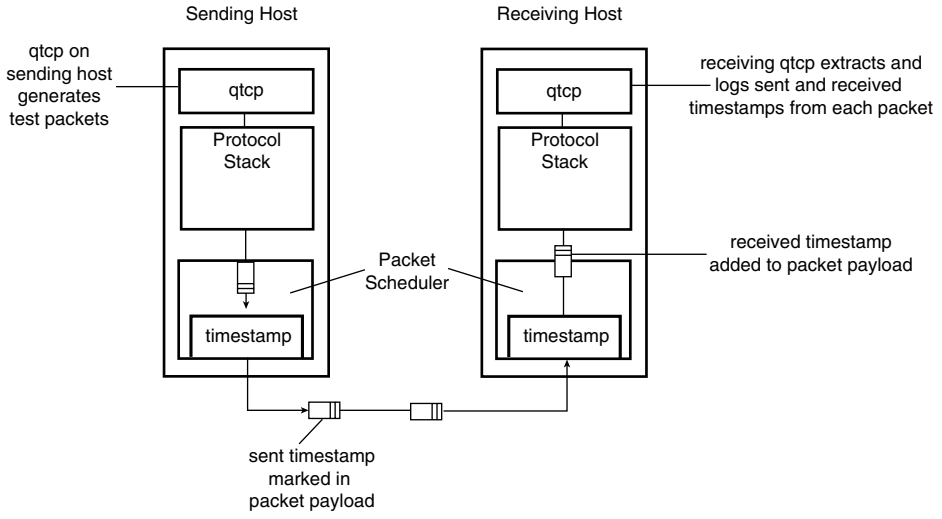
Even though qtcp invokes GQoS for the test traffic, the traffic will enjoy prioritized treatment only to the degree that QoS policies and admission control permit.

By default, the qtcp sender generates `PATH` messages requesting Guaranteed Service for the user-specified traffic profile (this will create a shape mode flow in the packet scheduler). The qtcp receiver generates a `RESV` message in response. Upon receipt of the `RESV` message, the sender begins submitting test buffers to the kernel for transmission to the network. Transmitted packets are marked with a DSCP and an 802.1p mark corresponding to the Guaranteed Service.

As packets are transmitted, the timestamp module on the sender stamps a sequence number and the local time (to a precision of 100ns) in each packet's payload. When packets are received, the timestamp module on the receiving host stamps the local time (per the receiving host) in each packet payload. Packets are passed up to the user-level qtcp process, where it generates a list of packet sequence numbers and sent/received timestamp pairs to be used in the post-processing phase.

Test stream mode is illustrated in Figure C.4.

Figure C.4 Test Stream Mode



Application Stream Mode

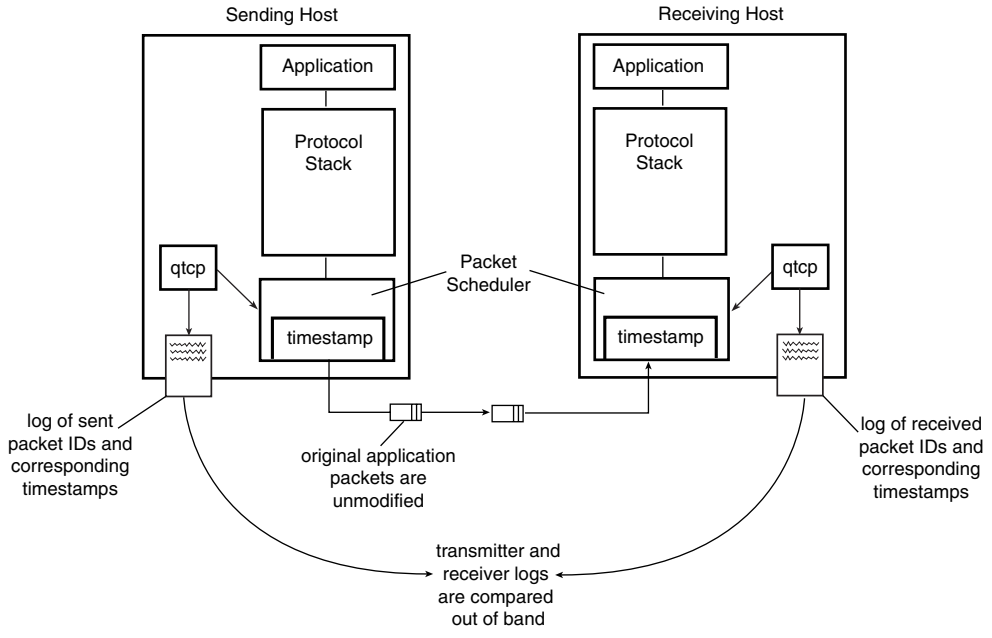
In application stream mode, a GUI is used to control the operation of qtcp. The GUI enables the user to select an existing application traffic stream to be measured. If the application of interest is QoS-enabled, then the corresponding traffic stream is selected from a list of flows that are established in the QoS traffic control components. If the application of interest is not QoS-enabled, the traffic stream to be measured can be selected from a list of existing TCP connections in the TCP/IP protocol stack. In this case, it is possible to request that qtcp invoke QoS on behalf of the application, using user-specified parameters.

Note

As noted previously (in the case of test stream mode), QoS will actually be *granted* only to the degree that policies and admission control permit it.

In application stream mode, the payloads of the packets under test are entirely consumed by the application under test. Thus, these cannot be used to carry timestamps. qtcp uses a proprietary out-of-band mechanism to generate the correct sent time/received time pairs. Application stream mode is illustrated in Figure C.5.

Figure C.5 Application Stream Mode



Termination of the Runtime Phase

In test stream mode, the user specifies the length of the test in terms of the number of packets to be transmitted (2,048 packets are sent by default). The runtime phase of the test terminates when the full number of packets has been sent. Alternatively, the user may terminate the test at any time from the command line.

In application stream mode, the user terminates the test at any time from the GUI.

Regardless of the operating mode, upon termination, the *receiver* qtcp process parses and processes the list of recorded timestamps. In test stream mode, three files are generated. These are named *filename*.raw, *filename*.log, and *filename*.sta (where *filename* is specified by the user). These include per-packet raw timestamps (or normalized latency measurements, depending on invocation options), clock skew adjusted latency measurements, and summary statistics, respectively. In application stream mode, qtcp generates four files, named TimeStmp.raw, TimeStmp.txt, Jitter.raw, and Jitter.log. The information contained in these files includes binary timestamp records and a text version of both the raw and processed timestamps.

The Role of the Packet Scheduler

The packet scheduler and its accompanying timestamp module are used to generate the sent and received timestamps from which latency measurements are computed. The advantage of generating timestamps in the packet scheduler is that timestamping is done as close as is practical to the time that each packet is actually transmitted to the network and received from the network. Thus, the latencies measured are indicative of *network* behavior rather than *operating system* behavior.

In certain cases, the packet scheduler is not or cannot be installed. In these cases, qtcp may still be used. However, the timestamping is then performed at the user level. Results based on user-level timestamping are subject to two sources of error when used to assess network behavior:

- Response time of the user-level process that performs the timestamping
- Variations in packet latencies through the network stack on sender and receiver

If the actual network latencies are relatively low, then these errors dominate and qtcp results describe operating system behavior rather than network behavior. If network latencies are relatively large, then network latencies dominate and qtcp results will describe network behavior.

Note

As a rule of thumb, it is safe to assume that the error introduced by relying on user-level timestamps is on the order of 15msec. This degree of error is very significant on LANs and, though less significant, cannot be ignored on most WANs either.

In addition to its role in generating timestamps, the QoS packet scheduler may be used to both shape and mark the traffic of interest on the sending host.

Post-Processing

At the completion of the runtime phase, qtcp obtains a list of sent timestamp/received timestamp pairs. These are processed during the post-processing phase to produce packet-by-packet latency measurements and summary statistics.

Obtaining Latency Measurements

To properly interpret the latency measurements generated by qtcp, it is helpful to understand how they are generated. This process is described in the following paragraphs. (Those readers not interested in a detailed discussion of how latency measurements are

generated may skip this section but should at least read the subsequent section "Considerations Affecting the Validity of qtcp Results.")

The timestamp pairs in the raw file generated by qtcp are based on the qtcp sender and receiver clocks. Although these clocks are expected to run at the same *rate* (or pretty close to it), they are not expected to be synchronized to a common time base. This poses special challenges in generating latency measurements. (See the accompanying sidebar titled "The Challenge of Clock Synchronization.")

The Challenge of Clock Synchronization

The challenge in obtaining accurate latency measurements is in synchronizing clocks between sender and receiver. If the sender and receiver clocks are perfectly synchronized, it is possible to obtain accurate latency measurements simply by subtracting the sent timestamp from the corresponding received timestamp. However, perfect clock synchronization is impossible to achieve. In most cases, host clocks are loosely set to the time of day in the local time zone. In this case, synchronization on the order of seconds or minutes can be expected. In special cases, synchronization software (such as Network Time Protocol, or *NTP*) can be used to achieve closer synchronization. Under ideal conditions, such software can achieve synchronization on the order of milliseconds or even sub-milliseconds. Unfortunately, this degree of synchronization requires both sender and receiver to be connected to a *time server* via a high-speed network. This requirement is often impractical.

Absolute Latency Is a Function of Clock Difference

The timestamps recorded by qtcp during the runtime phase are based on the local time as determined by the host on which they are recorded. Because there are no expectations regarding the synchronization of the clocks on the two hosts, it is impossible to determine the absolute transit latency of a packet. This problem is illustrated by the following equation:

$$(1)\ T_{rec} - T_{sent} = \Delta Clk + Lat_{abs}$$

Where the following is true:

T_{rec} is the received timestamp (in receiver's local time).

T_{sent} is the sent timestamp (in sender's local time).

ΔClk is the difference between the sender and receiver clocks.

Lat_{abs} is the absolute transit latency.

Simply put, this equation states that the difference between any pair of sending and receiving timestamps is equal to the sum of the time that it took for the corresponding packet to get from the sender to the receiver (the absolute transit latency) plus the difference

between the clocks on the sender and receiver. Thus, if the difference between the two clocks were known, it would be possible to calculate a packet's transit latency from the corresponding values of T_{rec} and T_{sent}—specifically:

(2) $Lat_{abs} = T_{rec} - T_{sent} - \Delta Clk$

If the sender and receiver clocks were perfectly synchronized, then ΔClk would be zero and the transit latency would simply be as follows:

(3) $Lat_{abs} = T_{rec} - T_{sent}$

However, because the sender and receiver clocks are not synchronized, ΔClk is finite and is unknown. Therefore, it is not possible to determine Lat_{abs} simply from T_{rec} and T_{sent}.

Deriving the Effective Clock Difference

Fortunately, it is possible to calculate an *effective* clock difference. This value *can* be used to determine meaningful (if not absolute) transit latencies from the timestamp pairs. To understand the significance and the use of the effective clock difference, it is first necessary to understand the components that comprise absolute transit latency.

Absolute transit latency is comprised of several subcomponents:

(4) $Lat_{abs} = Lat_{C} + Lat_{ser} + Lat_{pro} + Lat_{Q}$

Where the following is true:

- Lat_{C} is the latency due to speed of light (also known as propagation delay). This refers to the amount of time required for the electromagnetic signal representing a data packet to propagate through the network media (wired and/or wireless) between sender and receiver.
- Lat_{ser} is the latency due to serialization delay. It refers simply to the time it takes to transmit a packet of *N* bits over a medium of *M* bits/sec.
- Lat_{pro} is the latency due to fixed processing. This refers to the amount of time that it takes to perform any fixed processing that must be performed in network devices to move a packet from the sender to the receiver. For example, this would include the amount of time that it takes to copy a packet of a certain size from a device's input buffer to a device's output buffer.
- Lat_{Q} is the latency due to queuing delays. This is the latency that is incurred as a result of queuing under congestion in network devices.

From a QoS perspective, the first three components of latency are *fixed*—they remain constant regardless of the QoS mechanisms that may be applied. On the other hand, the fourth component, the latency due to queuing, is variable. It is entirely dependent on the QoS mechanism applied to the corresponding traffic flow.

So, the following is true:

(5) $Lat_{abs} = Lat_{fixed} + Lat_{Q}$

Where the following is true:

(6) $Lat_{fixed} = Lat_{C} + Lat_{ser} + Lat_{pro}$

From a QoS perspective, only the variable component of the absolute latency (Lat_{Q}) is interesting.

From (2) and (5), we get the following:

(7) $Lat_{Q} = T_{rec} - T_{sent} - (\Delta Clk + Lat_{fixed})$

We define the following:

(8) $\Delta Clk_{eff} = (\Delta Clk + Lat_{fixed})$

We substitute this:

(9) $Lat_{Q} = T_{rec} - T_{sent} - (\Delta Clk_{eff})$

Now, assume that at least one of the packets transmitted during a qtcp run experiences no queuing latency ($Lat_{Q} = 0$). Because equation [9] is true for each received timestamp/sent timestamp pair, it follows that for those packets that experience no queuing latency, the following is true:

(10) $\Delta Clk_{eff} = T_{rec} - T_{sent}$

And so, if it is possible to identify the timestamp pairs corresponding to those packets that experience no queuing latency, then it is trivial to calculate the effective clock difference (ΔClk_{eff}). From (8), the effective clock difference depends only on the fixed components of latency and on ΔClk, both of which are assumed to remain constant across all timestamp pairs (see the discussion of skew in the upcoming section "Compensation for Clock Skew"). Therefore, ΔClk_{eff} remains constant across *all* timestamp pairs. Consequently, once ΔClk_{eff} is known, it can be plugged back into equation (9) to determine Lat_{Q} for *each* timestamp pair.

Identifying the Timestamp Pair for Which Lat_Q=0

Equation (10) was based on the assumption that for at least one timestamp pair, $Lat_Q = 0$. From (9), for all timestamp pairs, the following holds true:

(11) $T_{rec} - T_{sent} = Lat_Q + (\Delta Clk_{eff})$

Because latency measurements in general (and Lat_Q, in particular) are always greater than or equal to zero, the timestamp pair(s) corresponding to a Lat_Q value of zero will be the pair(s) for which $T_{rec} - T_{sent}$ is least.

Note

It is possible that for some (or all) pairs, $T_{rec} < T_{sent}$. In this case, $T_{rec} - T_{sent}$ will be a negative number, and the pair corresponding to a Lat_Q value of zero will still be the one for which $T_{rec} - T_{sent}$ is the least number (for example, –10 is less than –4).

Algorithm for Computing Lat_Q for Each Timestamp Pair

Based on the algebra presented in the previous sections, qtcp uses the following algorithm to compute the per-packet, QoS-related latency for each timestamp pair:

1. Scan the list of timestamp pairs looking for the smallest value of $T_{rec} - T_{sent}$.
2. For the pair with the smallest value of $T_{rec} - T_{sent}$, compute ΔClk_{eff} using equation (10).
3. For all pairs, compute Lat_Q using equation (9).

This is referred to as the *normalization* algorithm. The application of this algorithm to the list of timestamp pairs is the most significant step in the computation of latency. Additional steps may be applied to compensate for clock skew and clock anomalies. These steps are described in subsequent sections titled “Compensation for Clock Skew” and “Compensation for Clock Anomalies.”

Example of the Normalization Algorithm

Consider the following example. Table C.1 lists three hypothetical timestamp pairs. (In real trials, a much greater number of timestamp pairs would be collected, and the timestamps would be more precise.)

Table C.1 Sample List of Timestamp Pairs

Pair Number	Sent Time	Received Time
1	12:01:10	12:07:13
2	12:01:11	12:07:19
3	12:01:12	12:07:18

For each pair, the difference between the received timestamp and the sent timestamp is computed, as shown in Table C.2.

Table C.2 Differences Between Sent and Received Timestamps

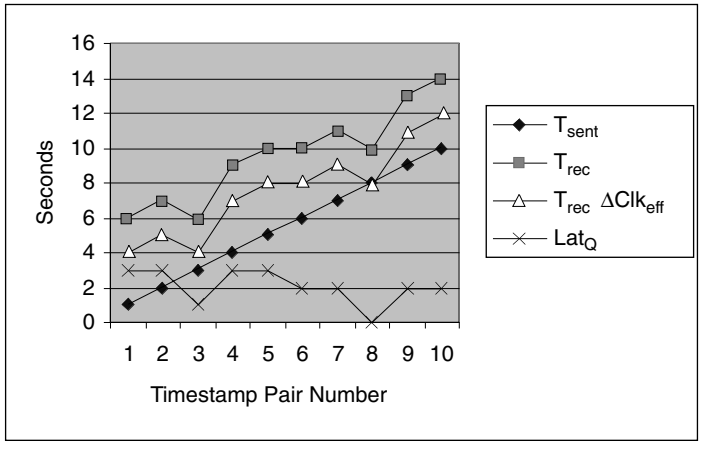
Pair Number	Sent Time	Received Time	Difference
1	12:01:10	12:07:13	6:03
2	12:01:11	12:07:19	6:08
3	12:01:12	12:07:18	6:06

The least difference is found to be 6:03, corresponding to pair number 1. This is therefore considered to be the effective clock difference. Using this value in equation (9) yields a QoS-related latency of 0, 5, and 1 seconds for the packets corresponding to timestamp pairs 1, 2, and 3, respectively.

Graphical Interpretation of the Normalization Algorithm

Figure C.6 illustrates the application of the normalization algorithm graphically.

Figure C.6 The Normalization Algorithm—Graphically



The curves labeled T_{rec} and T_{sent} represent the raw timestamp pairs. Pair number 8 can be seen to have the smallest value of $T_{rec} - T_{sent}$ (and therefore represents $Lat_Q = 0$). The value of $T_{rec} - T_{sent}$ for this pair is 2, which is therefore ΔClk_{eff}. The curve labeled $T_{rec} - \Delta Clk_{eff}$ represents an intermediate step in which T_{rec} is adjusted to remove the fixed latency component. Finally, the curve labeled Lat_Q yields the difference between $T_{rec} - \Delta Clk_{eff}$ and T_{sent} for each timestamp pair.

Compensation for Clock Skew

The normalization algorithm described in the previous section assumes that the sending and receiving clocks, while not synchronized, do run at the same rate. In other words, ΔClk is constant over time. However, the crystal oscillators that serve as the timebase for PC clocks are subject to slight deviation from their nominal frequency (on the order of several parts per million). This deviation is referred to as *clock skew*. Clock skew is a source of error in latency measurements. When measuring queuing latencies on the order of hundreds of milliseconds over a period of seconds or minutes, error due to clock skew is negligible. However, when measuring queuing delays that are much lower (such as on a high-speed LAN) or when measuring delays over a long period of time, error due to clock skew may become significant.

Recall from equation (5) that latency is considered to consist of a fixed component and a variable component. So far, the variable component of the latency has been assumed to result only from queuing latency. However, clock skew adds an additional (usually minor) component to the variable component of the latency, as follows:

$$(12)\ Lat_{var} = Lat_Q + Lat_{skew}$$

Usually this also is the case:

$$Lat_Q >> Lat_{skew}$$

Where the following is true:

Lat_{var} represents all variable latency components.

Lat_{skew} represents latency variation due to clock skew.

Fortunately, qtcp can estimate the variable component that is due to clock skew and can compensate for it.

To compensate for the clock skew, qtcp applies a skew-compensation algorithm in addition to the normalizing algorithm described previously. The skew-compensation algorithm works by recognizing that over time, Lat_{skew} can be expected to be either monotonically

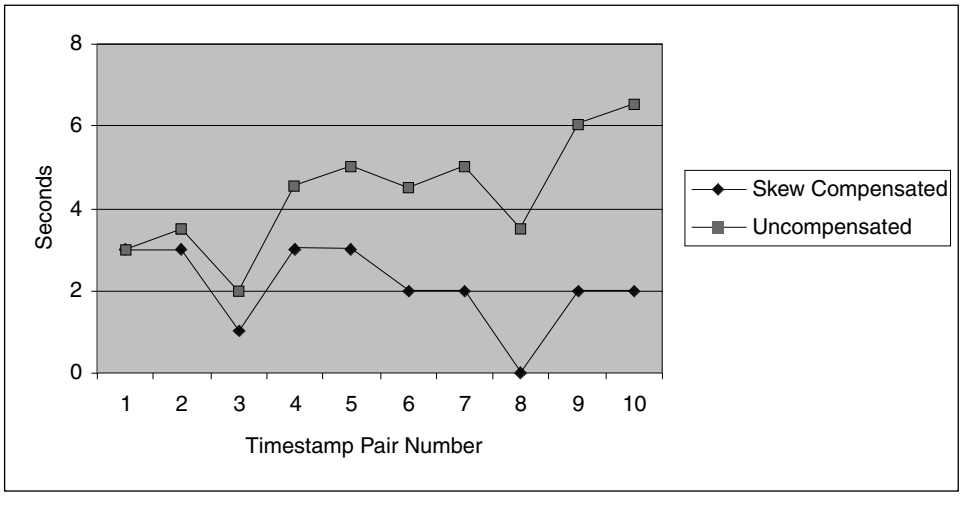
increasing or monotonically decreasing, while Lat_Q is expected to vary symmetrically around some constant mean. The first step in the skew-compensation algorithm is to compute the equation for a straight line that best fits the Lat_Q curve (illustrated in Figure C.6).

Note

qtcp uses one of two methods to determine the best-fit line. One method minimizes the sum of squares error to determine best-fit. This method assumes that latency is normally distributed. The other method minimizes absolute deviation as the goodness of fit measure. This method assumes that latency is distributed more like a double exponential about the mean.

Next, the slope of this line is calculated. If the slope of the line is zero, then over time, Lat_{var} is neither increasing nor decreasing, and there is no appreciable clock skew. If the slope of the line is finite, then there is clock skew. To compensate for the skew, all values of latency are adjusted by rotating the Lat_Q curve so that the slope of the best-fit line becomes zero. This is illustrated in Figure C.7.

Figure C.7 Clock Skew Compensation



In Figure C.7, the curve labeled "Uncompensated" represents latency measurements that include clock skew. Note that the curve has a general upward slope. Compensation removes this slope to produce the curve labeled "Skew Compensated."

Compensation for Clock Anomalies

On certain PC hardware platforms (generally older platforms using the piix4 timer chip), clock anomalies may be observed. In particular, the time may appear to periodically skip forward or backward by a specific (and relatively large) amount. qtcp can be invoked in a manner that attempts to compensate for this clock anomaly. Note that it may not detect all such clock anomalies, and even if it does detect them, it may not correct them properly in all occasions. However, it has been found to do so in most cases. Generally, newer PC hardware is immune to these problems.

If a plot of qtcp latencies produces a sawtooth pattern, it is likely that such clock anomalies are occurring and are not correctly being compensated for.

Considerations Affecting the Validity of qtcp Results

qtcp is a tool that assumes a model of network behavior and produces results based on these assumptions. The results must be considered in the context of the assumptions. Failure to do so could lead to incorrect conclusions. The previous section described the various algorithms that are applied to raw data to extract latency measurements, and the assumptions on which they are based. This section discusses the validity of those assumptions under various conditions.

Factors Affecting the Normalization Algorithm

Recall that the normalization algorithm is based on the assumption that at least one timestamp pair represents a packet that incurred no queuing latency. All remaining latency measurements for the same qtcp run are based on the difference between T_{sent} and T_{rec} for that timestamp pair. How valid is this assumption?

Queues in a healthy network device in a real production network tend to grow and shrink fairly frequently. If a qtcp run lasts for a sufficiently long time period, it becomes increasingly likely that at some point during the run, all or most of the network devices between sender and receiver are uncongested. If a sufficiently large number of packets are sent during the run, it becomes increasingly likely that one of the packets is sent during the period of noncongestion and therefore experiences no queuing latency.

It is hard to quantitatively determine the minimal length of run required and the number of packets that must be sent to obtain valid results. The following guidelines may be useful:

- An overloaded network may never be uncongested.
- Many production networks have idle periods (such as evenings on corporate networks). If the qtcp run can be made to span an idle period, one of the sent packets likely will experience no queuing latency.

- A healthy (not overloaded) production network will likely have brief idle periods quite frequently. Many network devices offer monitoring tools to check their utilization levels. These can be used to estimate the health of the network, device by device.
- In nonproduction networks (controlled networks), in which the user has control of network traffic, background network traffic should be suppressed for part of the qtcp run. A calibration mode is provided for this purpose (see the upcoming section titled "Calibration").
- Even if the network is not uncongested at some point during the qtcp run, latency measurements will not be useless. They will simply indicate latency variations above the lowest measured queuing latency, rather than above zero queuing latency.

Factors Affecting Clock Skew Compensation

As explained previously, clock skew compensation is based on the assumption that monotonically increasing or decreasing latency is attributable entirely to clock skew and should be filtered out. If the user is aware of and is attempting to measure monotonically increasing or decreasing latency attributable to factors other than clock skew, then the clock skew compensation processing should be suppressed (see the upcoming section "How to Use qtcp"). In this case, other means will have to be used to compensate for clock skew (if deemed significant).

Note

As a rule of thumb, it is possible to estimate the impact of clock skew on an uncompensated measurement by considering typical clock skews observed. Typical PC clock skew is on the order of 10 microseconds per second. Thus, a qtcp run of 100 seconds could result in a cumulative error of 1msec. An hour-long run could result in a cumulative error of more than 30msec, and a 24-hour run could result in a cumulative error of more than 800msec. Whether these errors are significant depends on the latencies being measured. An error of 1msec may not be significant when measuring transoceanic WAN latencies, but it is quite significant when measuring LAN latencies.

The clock skew compensation algorithm assumes a constant difference in rate between the sending and receiving clocks. However, PC oscillator frequencies vary slightly with temperature. Thus, if significant temperature changes occur during a qtcp run, the rate difference between the two clocks might not be constant. This could compromise the effectiveness of the compensation algorithm.

Note

When using qtcp's calibration mode (See the upcoming section titled "Calibration"), the clock skew is computed based on the calibration packets only. The resulting compensation factor is applied to all latency measurements.

Statistics

After the various normalization and compensation mechanisms have been applied to the complete set of timestamp pairs, qtcp computes summary statistics. This is the final step of the post-processing phase. Summary statistics include such data as the average and maximum queuing latencies measured. These are written to the .sta file.

How to Use qtcp

Although test stream mode and application stream mode use the same underlying measurement mechanism, the two differ significantly in terms of their usage. Test stream mode qtcp is command line-driven, while application stream mode is GUI-driven. The two are discussed separately in the following sections.

Note

Regardless of whether test stream mode or application stream mode is used, the QoS packet scheduler and the timestamp module should be installed. If these are not installed, qtcp will operate but will be limited in accuracy and in precision. See the previous section, "The Role of the Packet Scheduler."

The packet scheduler is not available on operating system versions predating Windows 2000. On Windows 2000, it must be explicitly installed. (See the section in Chapter 13 titled "The Ethernet Packet Scheduler" for installation instructions.) Another step is required to install the timestamp module.

Certain versions of the operating system after Windows 2000 include the packet scheduler and the timestamp module by default. On these versions it is not necessary to explicitly install the packet scheduler.

Test Stream Mode

In test stream mode, the user invokes qtcp from the command line on both a sending and a receiving host. The simplest command line invocation is as follows:

On sender: `qtcp -l64 -t 2.3.4.5`

On receiver: `qtcp -f"filename" -r`

The `-l` option on the sender specifies the buffer size to be used in the test. The `-t` option specifies that this is the qtcp sender. This option must be followed by the IP address of the receiving host. On the receiver, the `-f` option is used to specify the prefix of the report files that will be generated. The `-r` option indicates that this is the qtcp receiver.

Initially, the sender will print the message "Initiated QoS connection. Waiting for receiver." The receiver will print the message "Waiting for QoS sender to initiate QoS connection." At this time, the receiver and the sender are awaiting the required exchange of RSVP messages before beginning the data transfer. When the data transfer begins, the sender starts sending 64-byte buffers (as specified in the `-l` option on the sender) to kernel traffic control for transmission to the receiver at address 2.3.4.5. By default, kernel traffic control will pace transmitted packets to a rate of 100KBps.

qtcp prints a series of dots to the console, both on the receiver and on the sender. Each dot corresponds to 100 packets sent or received. Note that the first dot is printed on the receiver before the actual receipt of the first 100 packets. The dots should be used as an indication that qtcp is "alive."

Upon transmission of the specified number of packets (2,048, by default), the sender terminates with a message regarding the transmission rate. Note that the transmission rate and other statistics printed by the sender are from its perspective alone. That is, the rate that it prints is the rate at which it sends packets, and this does not necessarily correspond to the rate at which packets are received. Upon receipt of the required number of packets (or the termination packets) at the receiver, it terminates with the message "Received 2048 buffers." At this time, the receiving qtcp generates the files *filename*.sta, *filename*.raw, and *filename*.log.

Usage Variations

Table C.3 describes qtcp command line usage options and the corresponding parameters. (The third column indicates whether each option is useful on the qtcp sender [T], the qtcp receiver [R], or both [B].)

Table C.3 qtcp Usage Options

Option	Description		Default and Range
`-B`	This is the token bucket size. The token bucket size represents the largest burst that traffic control will transmit to the network. It should generally be set to be equal to the buffer size. (See the notes following this table.)	T	The default is 64 bytes.
`-m`	This is the minimum policed size advertised by RSVP signaling.	T	The default is buffer size.
`-R`	This is the token rate, the average rate at which data will be transmitted in kilobytes per second. This parameter may be used to control the interpacket gap on the sending host. (See the notes following this table.)	T	The default is 100KB. This should be lower than the media rate.
`-R##B`	Token rate, expressed in bytes per second.	T	See the `-R` entry above.
`-S`	This is the IntServ service type that will be signaled to the network and that will be used for local traffic control. (See the notes following this table.)	T	This is either GR (Guaranteed), or CL (Controlled Load).
`-e`	This option forces shaping to the token rate, regardless of the service type requested.	T	The default is shaping for GR only.
`-W`	This suppresses waiting for an RSVP reservation before transmitting the test stream. (See the notes following this table.)	B	This option takes no parameters.
`-v`	This option is used to set up an RSVP reservation only and then wait for the user to exit. No data is sent.	T	Data is sent by default.

Option	Description		Default and Range
`-f`	This option is used to specify the prefix name for the logging and statistics files on the receiver.	R	By default, no files will be generated.
`-n`	This is the number of buffers transmitted.	B	See the notes following this table.
`-c`	This is the number of calibration buffers to be sent.	B	See the notes following this table.
`-k#`	0: Do not compensate for clock skew. 1: Compensate for clock skew based on a linear regression, with goodness of fit measured by sum of squared error. 2: Compensate for clock skew based on a bracketing and bisection root finding method, with goodness of fit measured by sum of absolute error. 3: This is the same as 2, except also try to compensate for clock jumps (see the previous section, "Compensation for Clock Anomalies").	R	The default is 2.
`-y`	This option causes qtcp to proceed directly from the calibration phase to the measurement phase without prompting the user.	T	By default, qtcp prompts the user before proceeding from the calibration phase to the measurement phase.
`-p`	This is the destination UDP port number to which the qtcp session sends (on the receiving host). To run multiple simultaneous qtcp tests between the same pair of hosts, it is necessary to use separate ports for each test. This option may be used to force a specific port to be used. Otherwise, all streams will attempt to send to the default port on the receiver. This option must be used on both sender and receiver.	B	The default is port 5003.

continues

Table C.3 Continued

Option	Description		Default and Range
`-l`	This is the length of user-level buffers generated by qtcp. (See the notes following this table.)	T	The default is 1,472. This must be greater than 48 bytes.
`-d`	By default, dropped packets are omitted from the log files. If this option is specified, they are represented by dummy entries in the log files showing a sent and received timestamp of zero and the maximum latency experienced by any packet in the test run. This option can be used to accommodate different types of log file post processing.	R	The default is no dummy entries.
`-N`	This causes the raw file to be dumped *after* normalization.	R	The default is prenormalization. (See the notes following this table.)
`-M`	This is the `MaxSDUSize` specified in RSVP signaling messages.	T	The default is buffer size.
`-P`	This suppresses console reporting of dropped packets.	R	The default is to report.
`-u`	This reports user-mode timestamps in log files.	R	The default is kernel mode.
`-i`	Use especially compressible data in test packet payloads.	T	The default is less compressible.
`-q`	Log only every *n*th packet. For instance, `-q2` will log every other packet (0, 2, 4, and so on).	R	The default is `-q1` (log every packet).

Notes Regarding Usage Table: Relationship of Token Bucket Size, Buffer Size, MTU, and Packet Size

Certain combinations of token bucket size and buffer size should be avoided. These constraints are explained in the following paragraphs.

Applications transmit data by submitting *buffers* to the protocol stack on a sending interface. The protocol stack converts these buffers to *packets*. The packets are passed to the packet scheduler and then are transmitted to the network. Each interface has a *maximum transmission unit* (MTU) associated with the interface. This represents the maximum packet size that can be transmitted to the network. If the buffer submitted to the protocol stack is *larger* than the MTU size, then the protocol stack must *fragment* it into a sequence of packets that are smaller or equal to the MTU. (See the following sidebar titled "Fragmentation.")

Note

In actuality, the protocol stack adds a protocol *header* to the buffer contents in order to construct a packet. The header required for qtcp packets is 28 bytes in size (20 bytes of IP header + 8 bytes of UDP header). Thus, to avoid fragmentation, the buffer must actually be less than or equal to the MTU minus 28 bytes.

If the buffer submitted to the protocol stack is *smaller* than or equal to the MTU minus 28 bytes, then it will be sent as a single packet. As a result, the packets generated by the protocol stack are equal in size to the lesser of the MTU minus 28 bytes and the submitted buffer size.

Fragmentation

Fragmentation is undesirable for a number of reasons, as described here:

- Certain network QoS mechanisms are incapable of handling fragmented packets.
- The loss of a single packet results in loss of the entire corresponding buffer at the receiving host, even though all packets may have successfully traversed the network.
- The user-level qtcp process on the receiving host recognizes buffers, not individual packets. As a result, timestamps and sequence numbers would be recovered only from the first packet comprising each buffer. Information stamped by the sending timestamp module in the remaining packets of the buffer would not be recoverable.

Constraint 1: In order to avoid fragmentation, the buffer size submitted by the application must be smaller than or equal to the MTU size minus 28 bytes.

Since the packet size generated by the stack is the lesser of the MTU (minus 28 bytes) and the buffer size, if Constraint 1 is met, then the packet size generated by the protocol stack will be equal to the buffer size.

The packet scheduler on the interface is configured for a certain token bucket size. The token bucket size specifies the maximum amount of data that may be sent in a single burst. If a packet that is larger than the token bucket size is submitted to the packet scheduler for transmission, it will never be able to send the packet.

> **Constraint 2:** Packets submitted to the packet scheduler for transmission must always be less than or equal in size to the token bucket.

Because the packet size is equal to the buffer size, if Constraint 2 is to be met, the buffer size must always be less than or equal to the token bucket size.

If multiple qtcp buffers are transmitted in a single burst, the transmit timestamps in these buffers will be closely related. This tends to distort the results of the measurement. Optimal results are obtained when a single qtcp buffer (or packet) is sent at a time, with equal inter-buffer (inter-packet) gaps.

> **Constraint 3:** To prevent multiple qtcp buffers from being sent in a single burst, the buffers should be no smaller than the token bucket size.

Constraint 2 dictates that the buffer size must always be *less than* or equal to the token bucket size. Constraint 3 dictates that the buffer size should always be *greater than* or equal to the token bucket size. The only buffer size that can accommodate both constraints is the token bucket size. Therefore, the buffer size and the token bucket size should be equal. Furthermore, Constraint 1 dictates that both should be less than or equal to the MTU (minus 28 bytes).

Note

The default MTU size on LAN interfaces is typically 1,500 bytes. Thus the token-bucket size and the buffer size should be less than or equal to 1,472 bytes. This buffer size is determined by the `-l` option and by default is equal to 1,472 bytes.

Token Rate

As discussed previously, optimal results are obtained when the sending host sends packets with equal interpacket gaps. This behavior is ensured when the token bucket size is chosen to be equal to the buffer size. In this case, the token rate (determined by the `-R` option) determines the packet rate (and therefore the interpacket gap). For example, for a token bucket and buffer size of 64 bytes and a token rate of 16, packets will be sent at the uniform rate of 250 packets per second.

Each packet can be considered to be taking a snapshot of the network conditions at the time it is sent. Thus, the token rate can be used to select a sampling interval for the network under test. Too low of a sampling interval may cause transient network conditions to be missed. On the other hand, the token rate should be selected so that the packet rate is relatively low compared to the limitations of the measuring processes in the sending and receiving hosts. For example, the granularity of the timestamping clock in most PCs is on the order of 100nsec. At this granularity, packet rates above 10 million packets per second cannot be measured. For practical purposes, it is advisable that the packet rate be quite a bit lower than this.

The token rate selected should be lower than the media rate. When using qtcp to evaluate the effects of QoS on telephony traffic (for example), token rates on the order of 3–10KBps are recommended.

Service Type

The service type (selected by the `-s` option) can be used to select the IntServ service type requested by the RSVP signaling messages and to control the mode of the sending host's traffic control. The two types of service are GR (Guaranteed Service) and CL (Controlled Load Service). See Chapter 4, "Integrated Services," for a description of these services. Guaranteed Service is selected by default.

If there are RSVP/IntServ-aware devices in the network under test, then the choice of service type will affect the handling of the test traffic by these devices. Regardless of the existence of RSVP/IntServ-aware network devices, the choice of service will affect traffic control on the sending host by determining the mode in which the packet scheduler operates. Unless configured otherwise, the packet scheduler on the sending host operates in *shape* mode for Guaranteed Service and in *borrow* mode for Controlled Load Service. (See Chapter 13 for a discussion of these modes.) In shape mode, the packet scheduler shapes transmitted traffic to the token bucket parameters. In borrow mode, the packet scheduler does not shape traffic to the token bucket parameters. Instead, it transmits traffic up to the media rate, demoting in priority those packets that are transmitted in excess of the media rate. Note that if RSVP signaling requests are not denied in the network, the packet scheduler will also mark transmitted packets.

Operating qtcp in borrow mode tends to result in bursty traffic and is not recommended for measurement purposes. However, it may be desirable to select Controlled Load Service (as opposed to Guaranteed Service) to compare the effect of various service types when there are RSVP/IntServ-aware devices in the network under test. In this case, it is recommended that the packet scheduler be configured to operate in shape mode by specifying the `-e` flag when running qtcp.

No Wait Flag

By default, qtcp will not begin data transmission until an RSVP reservation is in place. The `-W` option allows data transmission to proceed even when there is no reservation in place. This option can be invoked to enable testing when the network under test prevents a reservation from being installed. This could happen for the following reasons:

1. A firewall in the path between sender and receiver is configured to block RSVP messages.
2. An RSVP-aware device in the network is rejecting the RSVP request because of lack of resources, policy, or other reasons.

Note that by using the `-W` option, the synchronization inherent in RSVP is lost. This means that the sender will not wait for the receiver to be started. Thus, when using this option, it is advisable to start the receiver *before* the sender.

Number of Buffers

The `-n` option can be used to select the number of buffers that the transmitter sends or that the receiver expects to receive during a test run.

Too small of a number of buffers may result in the error message "Time interval too short for valid measurement." Too large of a number of buffers on the receiver will result in the error message "Could not allocate X bytes for log buffer," indicating that the receiver was incapable of allocating sufficient memory to record-timing data for the number of buffers that it would have to receive.

Note that both sender and receiver use 2,048 as the default number of buffers. If the `-n` option is used on the sender to restrict the number of buffers sent to less than the default, then no action is required on the receiver. However, if the sender is configured to transmit more than the default, then the corresponding option must also be selected on the receiver to prevent it from terminating after the default number of buffers has been received.

It is possible to invoke qtcp for a particular *time duration* rather than a particular number of buffers. To do so, use the `s` suffix to the `n` parameter. To invoke a qtcp run of one hour (3,600 seconds), for example, use the option `-n3600s`.

Calibration

Recall that the computation of latency measurements is based on the assumption that at some point during the qtcp run, the network is uncongested. When the network under test is a controlled network, it is expected that the network is uncongested until the user purposely introduces noise. In this case, the calibration feature of qtcp enables the user to

invoke qtcp so that the qtcp run spans both an uncongested period and a congested period. As a result, both ΔClk_{eff} and clock skew can be computed with greater accuracy than would otherwise be possible, thereby improving the accuracy of the final post-processed results.

Note

The calibration option is not available for the application stream version of qtcp.

The `-c` option should be used to enforce a calibration phase. This option will cause qtcp to transmit a specified number of buffers upon invocation and then to pause, awaiting action from the user before continuing. The time between qtcp invocation and this pause is referred to as the calibration phase. qtcp assumes that the packets sent during the calibration phase reflect the characteristics of a quiescent network.

The argument to the `-c` option is the number of buffers to be transmitted *during the calibration phase*. On the sender, these are sent *in addition* to the number of buffers specified by the `-n` parameter (or the default of 2,048).

When the `-c` option is used, computations used to compensate for clock skew (see the previous section "Compensation for Clock Skew") are based *only* on the timestamp pairs collected during the calibration phase. Other statistics are based on all timestamp pairs.

To take advantage of the calibration feature, the user should begin the qtcp run on a quiescent network, using the `-c` option. After the specified number of calibration buffers have been sent, qtcp will pause and will prompt the user with the message "Calibration complete. Type 'c' to continue." At this time, the user should start any noise-generating tools being used to congest the network under test. After noise generation has been started, the user should type **c** to continue with the measurement phase of the test.

The following sample invocation may be used:

On sender: `qtcp -c1000 -n1000 -l64 -t 2.3.4.5`

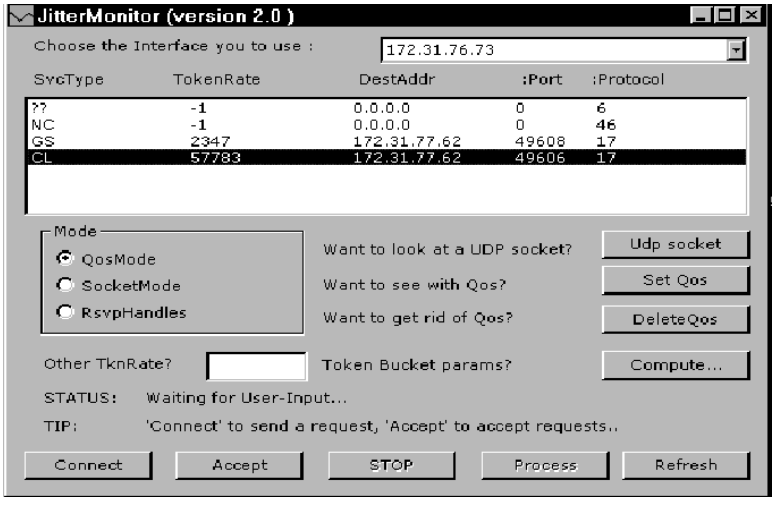
On receiver: `qtcp -c1000 -f"filename" -r`

This will cause the sender to transmit 1,000 calibration buffers, followed by 1,000 noncalibration buffers. The receiver will use the first 1,000 buffers received to normalize for clock skew.

Application Stream Mode

Usage of qtcp in application stream mode is entirely via the dialog box illustrated in Figure C.8.

Figure C.8 Application Stream Mode GUI



Note

The dialog box is titled Jitter Monitor. This is another name for application stream mode qtcp.

Invoking Jitter Monitor

The Jitter Monitor GUI should be invoked on both the sending and receiving host by typing **jittermon<cr>**. The dialog box illustrated in Figure C.8 will appear. Jitter Monitor initializes in accept mode. This means that it is waiting for a peer instance on a remote host to initiate the Jitter Monitor run. To initiate a run, the user must press the *Connect* button on one of the Jitter Monitor peers. Before doing so, however, it is necessary to select an interface to be used and a traffic flow to be monitored.

Note

Jitter Monitor automatically selects an interface to be used. If there is only a single interface on the host on which Jitter Monitor is running, the corresponding interface address will appear in the pull-down menu at the top of the dialog box, and this interface will be automatically selected. If there are multiple interfaces, the user may select an alternate interface from this menu.

Selecting a Traffic Flow to be Monitored

Traffic flows may be selected for measurement using one of two mechanisms, depending on the submode for which Jitter Monitor is configured. The user uses the first two radio buttons in the Mode area of the dialog box to select a submode.

The QoS Submode

This is the default mode. This mode is used to measure the behavior of traffic flows that have previously been explicitly created using specific QoS mechanisms. These flows may have been created in response to QoS-enabled applications that make use of the GQoS API or, alternatively, may have been created by a third-party traffic-management application via the traffic control API. (See Chapter 11, "The Microsoft QoS Components," for a discussion of the GQoS API and the traffic control API.)

In either case, the QoS-enabled traffic flows are displayed in the Jitter Monitor dialog box. The user may select any of the displayed traffic flows for measurement. Note that Jitter Monitor will expect to find a peer instance on the host with the destination address specified in the selected traffic flow. If there is no Jitter Monitor instance running on this host, Jitter Monitor will fail to execute a test run.

The Non-QoS Submode

Jitter Monitor is configured for this mode by selecting the *SocketMode* radio button. This mode is used to measure the behavior of traffic flows that have not been explicitly created using a QoS mechanism. As such, this mode is particularly useful for measuring the behavior of traffic flows corresponding to non-QoS-enabled applications. In the non-QoS mode, the Jitter Monitor dialog box displays all TCP traffic flows present in the TCP/IP network stack. (At the time of this writing, UDP flows cannot be measured in this mode.) The user may select any of the displayed traffic flows for measurement. As is the case with the QoS submode, a peer instance of Jitter Monitor must be running on the host with the destination address specified in the selected traffic flow.

An additional feature of the non-QoS submode is the capability to invoke QoS on behalf of the traffic flow of interest. This feature is discussed in the upcoming section "Invoking QoS for Non-QoS-Enabled Applications."

Starting and Stopping the Jitter Monitor Run

When an interface and a traffic flow are selected, the user can initiate the Jitter Monitor test run. To do so, the user presses the *Connect* button on the dialog box on the sending host.

The test run can be terminated by pressing either the *Stop* button or the *Process* button. The *Stop* button immediately terminates the test run without generating results. The *Process* button terminates the run, applies post-processing, and generates the results files described previously.

Invoking QoS for Non-QoS-Enabled Applications

An attractive feature of the non-QoS submode is the option to invoke QoS on behalf of a selected traffic flow that corresponds to a non-QoS-enabled application. Using this feature, the user is able to investigate the benefits of invoking QoS for the traffic flow of interest.

To do so, the user presses the *Set QoS* button. This brings up a dialog box that prompts the user for the QoS parameters to be used when invoking QoS. These include token bucket parameters as well as the service type to be used.

Note

The user may use the *Compute* button to help determine the token bucket parameters to be specified. (See the next section titled "Computing QoS Parameters.")

When the user has specified the QoS parameters to be used for the Jitter Monitor run, the *Connect* button should be pressed to start the run. Jitter Monitor automatically invokes QoS on behalf of the application, using the specified QoS parameters.

Computing QoS Parameters

When using Jitter Monitor to invoke QoS on behalf of non-QoS-enabled applications, it is necessary to specify the QoS parameters to be used. Selection of the service type is relatively straightforward (see recommendations in Chapter 4 and Chapter 12). However, specification of the appropriate token bucket parameters could be quite difficult for certain applications.

Note

Token bucket parameters are ignored if Null Service (Qualitative Service) is selected.

For this reason, Jitter Monitor includes a feature that can be used to compute a set of token bucket parameters appropriate for the traffic flow of interest. To use this feature, a *characterization* run is necessary. The user initiates a characterization run by pressing the *Compute* button while the application is generating traffic on the flow of interest. During the characterization run, Jitter Monitor examines the profile of traffic transmitted on the flow. The characterization run is terminated by pressing the *Process* button. At this time, Jitter Monitor generates a text file containing recommended token bucket parameters.

Note

There is not necessarily a single token bucket size and a single token rate that characterize the measured traffic flow. Instead, any number of bucket size/token rate pairs can characterize the flow. The file generated by Jitter Monitor will offer multiple bucket size/token rate pairs.

The user is expected to select a set of operating parameters from the output generated by Jitter Monitor at the end of the characterization run. These parameters can then be used to control QoS invocation for a subsequent Jitter Monitor measurement run.

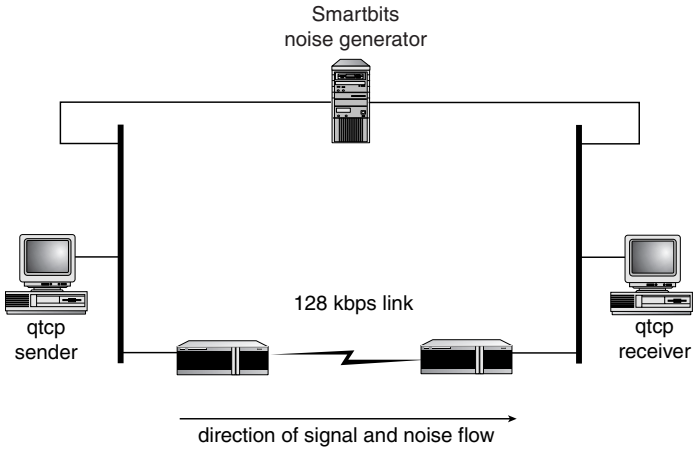
Disabling QoS Invocation

When the user elects to set QoS for an application, Jitter Monitor creates a QoS flow on behalf of the application. This flow can be referenced by its RSVP handle. To *disable* QoS on the selected flow, the user should first select the *RSVP Handle* radio button. Jitter Monitor then displays the QoS flow that has been created. QoS can be disabled for the selected flow by pressing the *Delete QoS* button.

Sample Results

This section presents sample results generated by test stream-mode qtcp runs. Results are presented for two measurement runs. Both runs were executed on a controlled network consisting of two routers connected by a 128Kbps serial line. Each router was also equipped with an Ethernet interface. The qtcp sender was connected to one of the router's Ethernet interfaces. The qtcp receiver was connected to the other. The sample configuration is illustrated in Figure C.9.

Figure C.9 Sample Configuration



Note the Smartbits noise generator in addition to the sending and receiving hosts. The sending port of the noise generator was connected to the same Ethernet network as the qtcp sender. The receiving port of the noise generator was connected to the same Ethernet network as the qtcp receiver. The noise generator was programmed to send 100Kbps of noise traffic from sending port to receiving port. Thus, the noise traffic generated competed against the qtcp test traffic for resources on the 128Kbps link. The noise generator was programmed to generate a mix of packet sizes that simulates the typical load on an enterprise WAN link.

qtcp was invoked in a manner intended to simulate a telephony traffic flow. The following parameters were used:

Sender: `qtcp -l64 -R3 -c1000 -n1000 -t 2.2.2.2`

Receiver: `qtcp -f"test" -c1000 -r`

This results in a test traffic flow of 64-byte packets sent at a rate of 24Kbps.

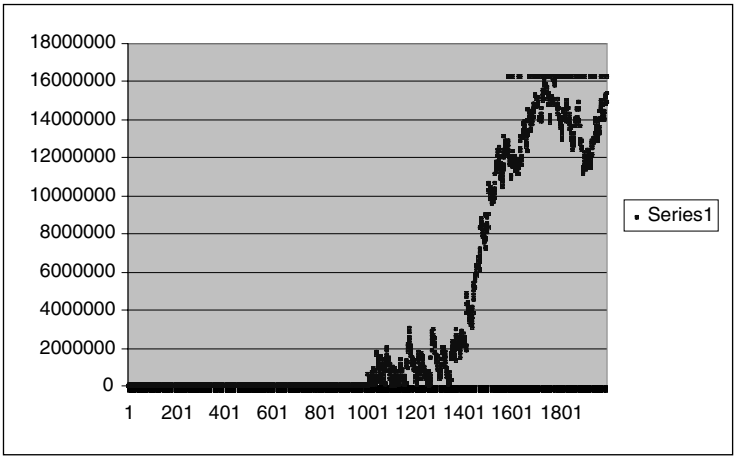
The purpose of the test runs illustrated was to examine the utility of RSVP in protecting the signal flow on a WAN link driven to near saturation. The first run was a control run and was executed with RSVP disabled on the routers. For the second run, both routers were configured for standard RSVP/IntServ functionality with weighted fair queuing enabled on the WAN interfaces. In both runs, RSVP signaling was generated by the qtcp sender and receiver.

The results of the two runs are illustrated in Figures C.10 and C.11. Plots were generated from the .log files that are created by qtcp. In both figures, the x-axis represents the sequence number of the timestamp pairs, and the y-axis represents the normalized (and clock skew compensated) latency in units of 100nsec.

qtcp Run Without RSVP Enabled

Figure C.10 illustrates the results of the test run for which RSVP was not enabled in the routers. The plot illustrates the queuing latencies experienced by the signal packets without RSVP protection. Note that the first 1,000 packets were sent during the quiescent calibration phase. As a result, these show negligible latency. The second 1,000 packets, however, were sent while background noise was generated. These show a steadily increasing latency, up to more than 1.6 seconds. Also, note the set of points aligned horizontally near the top of the plot. These correspond to dropped packets (which are represented in the .log file by the maximum latency measured during the run).

Figure C.10 Results with No RSVP Protection



qtcp Run with RSVP Enabled

Figure C.11 illustrates the results of the test run for which RSVP was enabled. The plot illustrates the latency experienced by the signal packets with RSVP protection. Again, the first 1,000 calibration packets show very low latency (note that the y-axis scale in Figure C.10 is adjusted to show latencies more than 10 times greater than the y-axis scale in Figure C.11). The second 1,000 packets show a distribution of latencies. However, unlike the plot in Figure C.10 (in which queuing latencies reached 1.6 seconds), the maximum queuing latency in this case is limited to 100msec. Furthermore, no packets were dropped.

The 100msec latency bound is not surprising. This is the amount of time that it takes to send a 1,500 byte packet on a 128Kbps link.

Figure C.11 Results with RSVP Protection