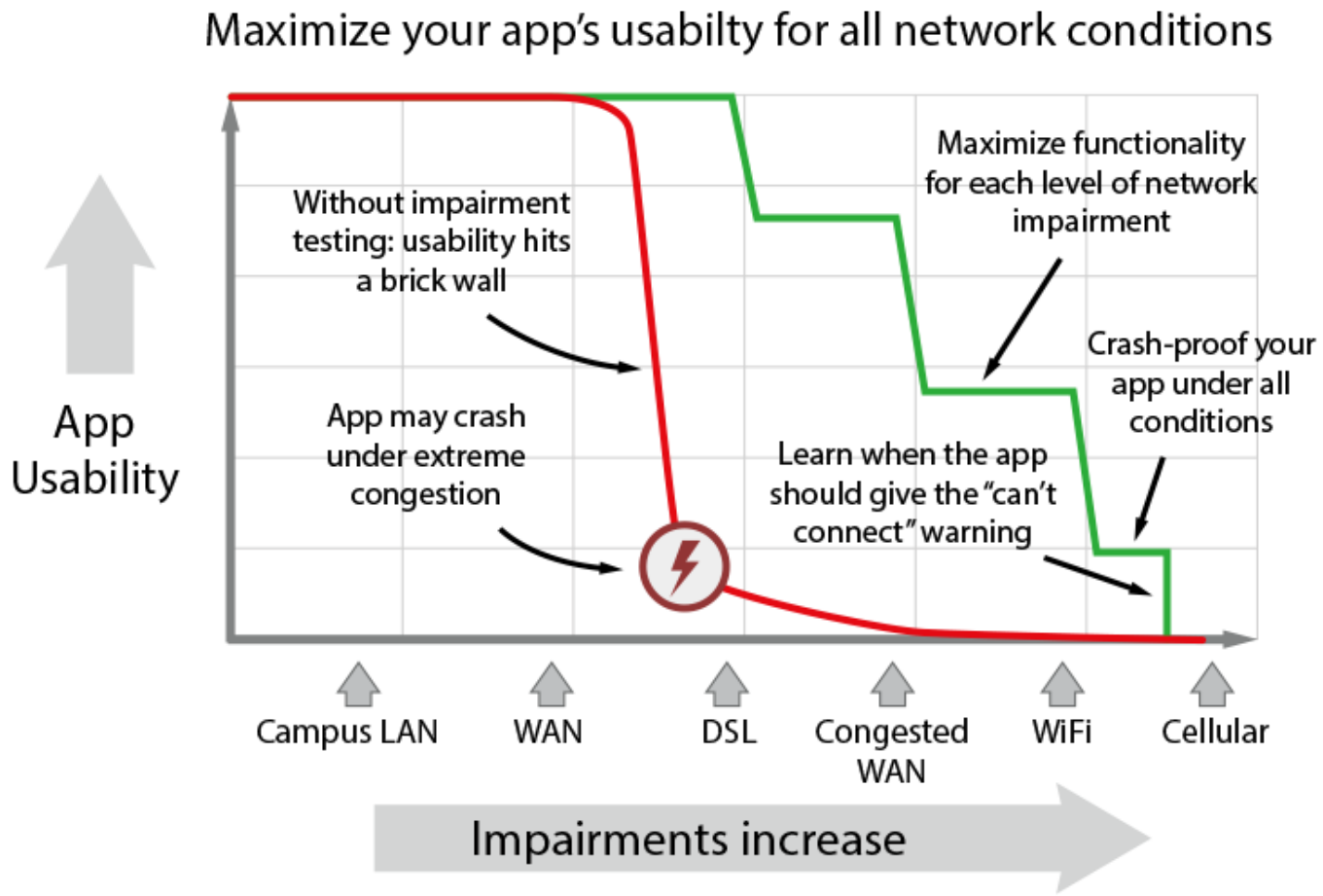
Testing Apps with Network Impairment Emulators

1 Testing Apps for the Network

1.1 Expanding the usability curve

Web and mobile developers need to test their application in the face of different network impairments, like limited bandwidth, additional round trip delay, and packet loss. Without impairment testing, app usability typically hits a brick wall at a certain critical point of reduced bandwidth, or excess latency, and then possibly crashes under severe congestion:



Apps require testing under higher network impairments, to push the usability curve further into the impairment zone. Developers use impairment testing to tune their apps in several ways:

- App developers can learn the limits of what the app can do at each impairment level, and

provide a subset of functionality that works within that scenario of lower bandwidth, or higher latency / jitter.

- The app should avoid crashing, even under the stress of worst-case network congestion.
- App developers can determine the appropriate congestion level at which the app should give up, and display the "can't connect" message.

1.2 App test criteria

Developers can use network impairment emulators to replicate real-world network problems, in order to test worst-case scenarios that apps might encounter. Different network emulators offer very different levels of test coverage and ease of use. In this paper, we'll consider these emulators:

Type of impairment tool	Product	Ease of use	Web app test coverage	Mobile app test coverage
Browser plugin	Chrome DevTools	**High**	**Low**	**N/A**
PC / Mac application	Apple Network Link Conditioner	**High**	**Low**	**N/A**
Layer-3 Netem gateway	Facebook ATC	**Low**	**Medium**	**Medium**
Layer-3 Netem gateway	Linux Netem	**Very Low**	**Medium-High**	**Medium-High**
Layer-2 Dedicated hardware	InterWorking Labs KMAX	**High**	**High**	**High**
Physical Device farm	AWS Device Farm	**N/A**	**N/A**	**N/A**

First, some critical terminology: A **network emulator** is different from a **device emulator**. When you build a mobile app, you will likely test it out on a virtual device emulator that runs on the same PC as your app development tools. Running the app on a device emulator is usually a lot quicker and more convenient that uploading your app to a physical device. But the device emulator runs the app with no stress testing; in fact, the emulator can run your code faster than the actual device; it has no provision for impairing network traffic flowing in/out of the app. An example a "device emulator" is the AWS Device Farm. The AWS Device Farm lets you test your app on a large number of real, physical devices installed in the data center; these devices are not exactly emulators, because a real mobile device is involved; but this approach is often referred to as the "AWS emulator". It's important to realize that whether you are using a device emulator on your development PC or in the cloud, the emulator doesn't add network impairments for testing: you'll need a separate network emulator, in addition to your device emulator.

Developers can use several types of impairment tools, depending on where the impairment is located:

- External tools that operate in the network path between an application and a server. This paper examines several such tools: Linux Netem, KMAX from InterWorking Labs, and the Facebook ATC (Augmented Traffic Controller) impairment gateway.
- Client-side tools that run at the network stack layer. This paper covers the Apple Network Link Conditioner.
- App-embedded tools that operate within the app framework itself. We'll take a look at the Network Throttling feature on the Chrome browser.

This paper covers three types of apps:

- Browser-hosted web apps, running on the desktop or a smart phone, and constrained to the browser sandbox.
- Desktop web apps, such as Chrome Web apps, that run outside the browser, and have access to a larger sandbox.
- Stand-alone mobile apps running on a smart phone.

While all types of apps can establish a TCP connection to the server, one crucial difference is that Desktop web apps, and stand-alone mobile apps, can establish TCP and UDP connections to arbitrary IP addresses, for things like content streaming, or custom server APIs. Because UDP has no error recovery or re-ordering mechanism, apps utilizing UDP connections must handle worst-case packet loss / corruption / delay / re-ordering.

2 Impairment tests

Your app has several layers of network connectivity between it and the server:

- Application level packet handling
- TCP or UDP stack
- Operating system housekeeping
- Local network infrastructure, like the local gateway and DNS
- Network backbone infrastructure to your server

To test your app, you'll need tools that can emulate network impairments at all these layers.

2.1 TCP-centric tests

Apps that rely on TCP require nearly the full set of impairments for comprehensive testing:

Rate limiting

In order to fully characterize a low-bandwidth link, the impairment process should emulate several attributes:

- Low-bandwidth link emulation, which emulates the bit-for-bit clocking on a WAN access link.

- WAN access link input queue packet drop
- WAN link packet overhead emulation: Some WAN links, such as cellular connections, have low efficiency due to small packets sizes and/or packet overhead.
- Different queue management algorithms common in WAN access points, such as RED (random early detection).

Delay

For apps that rely on TCP, It's necessary to emulate a constant fixed delay, plus a stochastic jitter delay:

- Extending the fixed delay with an impairment emulator stresses the round-trip time (RTT) of packets between your app and the server. If the RTT exceeds 100 ms, users will perceive the user interface as sluggish.
- The jitter delay embodies the variation in packet reception times as a result of the TCP stack handling corrupt, dropped, duplicated, or out-of-order packets. Jitter tends to stress the packet receiver code in your app.

Packet drop, corruption, duplication

Even though the TCP stack handles packet drop and corruption, increasing these impairments beyond a certain critical level will cause the TCP rate control feedback loop to drop down to a much lower bitrate, and the packet latency will go up. It's important for the app to adapt to a new steady-state bitrate and delay in the presence of a flaky network.

For packet corruption impairments, two features allow better testing:

- The ability to set per-bit corruption probabilities instead of per-packet corruption probabilities: this type of corruption is more realistic.
- An option to only corrupt bits in the Ethernet payload, not the Ethernet header: this option makes it much easier to debug problems. Some diagnostic tools like Wireshark do not work well if the Ethernet headers are corrupted.

2.2 UDP-centric tests

Apps primarily use UDP streams for real-time audio and video content, such as audio/video conferencing. If a packet becomes unusable due to corruption, drop, or latency, then it's too late to ask the sender to re-transmit the packet: instead, the receiver uses concealment techniques to try to minimize the effects of the unusable packet.

In addition to **Rate limiting** and **Delay** impairments, several additional impairments are essential for apps that utilize UDP:

Resequencing

This impairment accumulates a prefixed number of packets, then releases them all at once. This impairment can approximate a series of congested network queues between the app and the server. Resequencing is more advanced than just reordering packets; it implements a queue-

and-forward process.

Packet drop, corruption, duplication

If your app is consuming UDP streams, it needs to handle extreme cases of these impairments without crashing, and likely needs some kind of concealment algorithm in the presence of sustained packet drop/corruption.

2.3 Other network protocols

DHCP

When the smart phone turns on, if your app was sleeping in the background, it might get pulled into focus, and it needs to start showing content. DHCP determines how fast the phone can get an IP address. A flaky wireless connection can delay a DHCP offer, and your app needs to figure out how to look lively in the absence of an IP address. To test this situation, you'll need to emulate **packet drop** of DHCP packets, to extend the address request delay.

Layer-2 ARP

The operating system invokes Address Resolution Protocol under the covers when your app establishes the first connection out the gateway. You can test ARP problems by classifying the layer-2 ARP packets; then applying a high **packet drop** percentage to just those packets.

DNS

If your app uses anything other than hard-wired IP addresses, your app will suffer an initial delay when it resolves a hostname using DNS. In addition to extra delays from your app to the nearest DNS server, your app needs to plan for delays between that server and other authoritative servers. A complete impairment test tool can classify DNS packets, then add worst-case **packet delay**.

3 Impairment capabilities

For testing apps, a network emulator needs more than just basic impairment features.

3.1 Flexible packet classification

Selecting the right packets

For testing apps, you need a packet classifier that can route any packet (not just layer 3 TCP packets) to different sets of impairments, based on the type or contents of the packet. This capability allows you to replicate real-world situations, and to isolate impairments to track down bugs in your app.

You'll want to create packet classifications for each of these types:

- Layer 2 packets like ARP request/response
- VLAN tag
- Low-level network connectivity, like DHCP and DNS
- Different TCP addresses/ports
- TCP packets vs UDP packets
- Multicast vs unicast packets
- Smaller sized packets (used for audio streaming), vs larger packet.
- Packets tagged with higher vs lower QoS
- Packets containing certain byte patterns

Isolating traffic

In your test lab, multiple packets steams will likely pass through your impairment emulator, including packets from multiple endpoints, plus packets forwarded by any kind of network gear. It is essential to use flexible packet classification to isolate, and impair, only those packets that require emulated impairments; these are the packets to/from your app that will flow through a real-world WAN when a user fires up your app. In particular, there are three categories of packets that you'll want to **exclude** from the impairments:

- Packets from your app that won't be going through the WAN.
- Packets from other sources not related to your application
- Debug/Control connections to your app, as part of your test harness work flow. If your app is remotely controlled/configured, as part of an automated test system, you don't want this control mechanism to be impaired by the emulator.

3.2 Flexible impairments

The more flexibility you have with the impairments, the easier it is to test more scenarios and corner cases.

Asymmetric configuration

The connection between your app and the server is almost always asymmetric. Download speeds typically have a higher bitrate that upload speeds. For any impairment, a network emulator should allow you to set independent controls for the upload and download directions.

Burst mode

In the real world, impairments can suffer sustained bursts; once an impairment becomes active, it can be "sticky", and persist for a few packets in a row. It's important to be able to turn on this stochastic burst behavior, because that's what real networks do. It's also essential to be able to specify a "very sticky" burst behavior; in this case, an impairment rarely happens; but when it does, the impairment is sustained continuously for a large number of packets before the packet

stream recovers.

Easy A/B testing

The emulator should have an easily accessible on/off button, preferably hooked up to a hot key or mouse button, so you can quickly and easily observe your app with and without the impairments, and compare the operation.

Waveform specification

Impairments change over time, with time constants anywhere from seconds to days. Your app needs to adapt to these long-term changes. A network impairment engine can add significant value if it is possible to quickly specify a time-varying waveform expression for each impairment parameter, without writing code to an API. Waveform expressions are the fastest way to implement automation.

Scripting

An impairment emulator that exposes a control API can be integrated with a test infrastructure. API scripting is particularly useful for two categories of testing:

- Performing exhaustive overnight tests.
- It is often very useful to run an impairment (or a collection of impairments) through a pattern of changes. For example, a simple TCP stress test is to slowly ramp-up the end-to-end latency in one direction over a period of a minute or two and then to suddenly drop that latency back to a low baseline, and then begin the cycle anew. That kind of patterned change can put a lot of stress onto a TCP stack's congestion detection and recovery code.

Transitional control

There are three possible buffering mechanisms in an impairment engine:

- The input queue to a rate limiter
- The input queue to a delay module
- The input queue to a resequencer

When using either a waveform expression or automated scripting, it's essential to be able to control what happens to the packets in these queues when the modules are enabled/disabled, or when parameters change; there is a big difference between dropping all packets in a queue, vs forwarding all packets in a single burst. The desired options include:

- Dropping all packets
- Forwarding all packets immediately
- Sending packets at the originally scheduled times, even though the module is disabled.

Accuracy

Impairment accuracy is important for several reasons:

- I helps narrow down the exact tipping points when the app starts to fail. An app might work reasonably well with 5% packet loss, then seem to hit a wall at 5.1% packet loss.
- Especially for apps that use UDP streaming, there is often a trade-off between performance and resilience; an app might be able to provide better real-time performance with a small input buffer, but that small buffer will make it more susceptible to packet jitter. It's handy to determine exactly how an app behaves as a function of precise impairments.
- An impairment engine needs to retain accuracy, even in the face of high data rates that can stress the underlying CPU. If the impairment engine applies delay or jitter that varies as a function of data rate, the test engineers are going to see false positives.

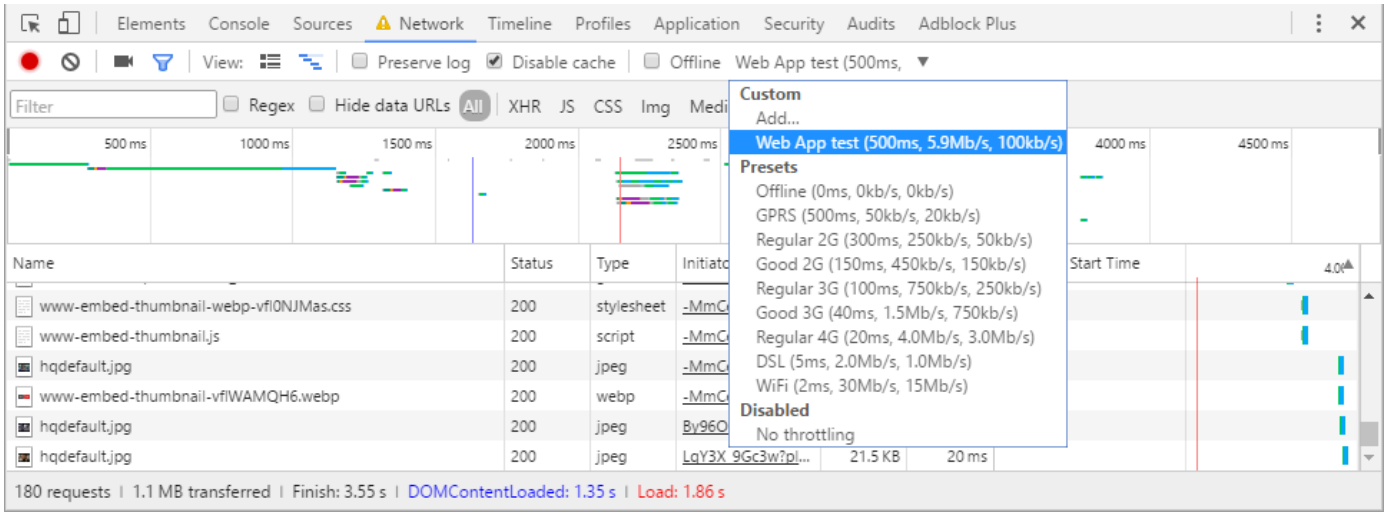
Impairment accuracy is higher on dedicated hardware platforms.

3.3 Ease of installation, configuration, and use

For ease of use, the ideal system requires zero installation time, no dependencies on any operating system, seamless insertion into the network path, and a flexible yet intuitive user interface for impairments, classification, and waveform expression. Easy integration with your test harness is essential.

4 Google Chrome DevTools

The Google Chrome browser offers network impairment emulation via a Developer Tools pane. To open the pane, select the Chrome main menu , then ***More Tools -> Developer tools***. In the DevTools pane, one of the top level tabs is ***Network***. In this tab, there is a toolbar item to select from a list of presets to throttle traffic to/from the web app running in the current browser tab:



You can select a standard preset, or define a custom impairment. Each entry has three impairment settings:

- Uplink bitrate
- Downlink bitrate
- Latency, applied to the uplink and downlink traffic

The impairments apply only to the current browser tab. A newly created tab will have no throttling, until you configure the impairments in the DevTools pane for that tab.

Pros

The impairment mechanism is conveniently built into the Chrome browser.

It's relatively easy to turn the impairment on/off using the drop-down control.

Cons

The impairment mechanism is missing several essential settings:

- Rate limiter queue drop
- Rate limiter WAN packet overhead
- Jitter

- Packet drop
- Packet corruption
- Packet duplication
- Packet resequencing.

And several impairment capabilities are missing:

- Burst mode
- Waveform specification
- API scripting
- Asynchronous delay setting
- Per-bit corruption probabilities
- Transitional control
- Impairment accuracy: For high data rates, a software-based browser tool may not be suitable.

The impairment mechanism has no packet classification:

- The impairments apply to all HTTP packets between the web app and the server. Web apps typically download content from a single web site, but often download JavaScript from multiple servers.
- The impairments cannot be customized per-server.
- Only the HTTP traffic can be impaired: the DNS / DHCP / ARP traffic is not affected. Impairing only the HTTP traffic results in unrealistic testing.

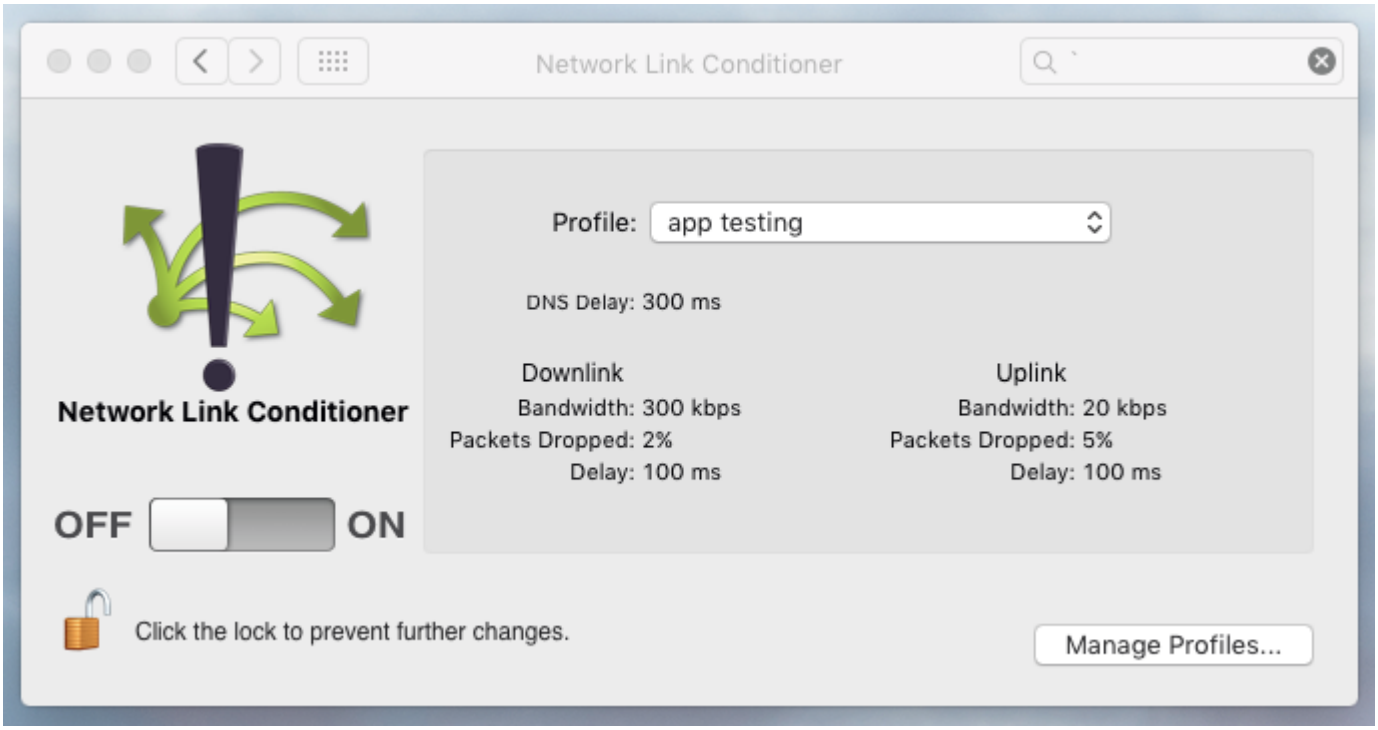
The ease of use is hampered by the lack of flexibility:

- If your web app launches a new tab, the impairments can't be applied automatically in the new tab: A throttling preset must be selected manually. Some web apps launch new tabs to show additional content, or as a way of linking to other sites.
- The impairment must be configured every time the browser is re-launched, or every time a new tab is launched.
- The impairments apply only to in-browser web apps, not to stand-alone applications, such as Chrome web apps that are available on the Chrome web app store.
- And finally, the capability is offered only in the Chrome Browser; there is no comparable functionality in either the Firefox Browser console, or in Firefox Extensions.

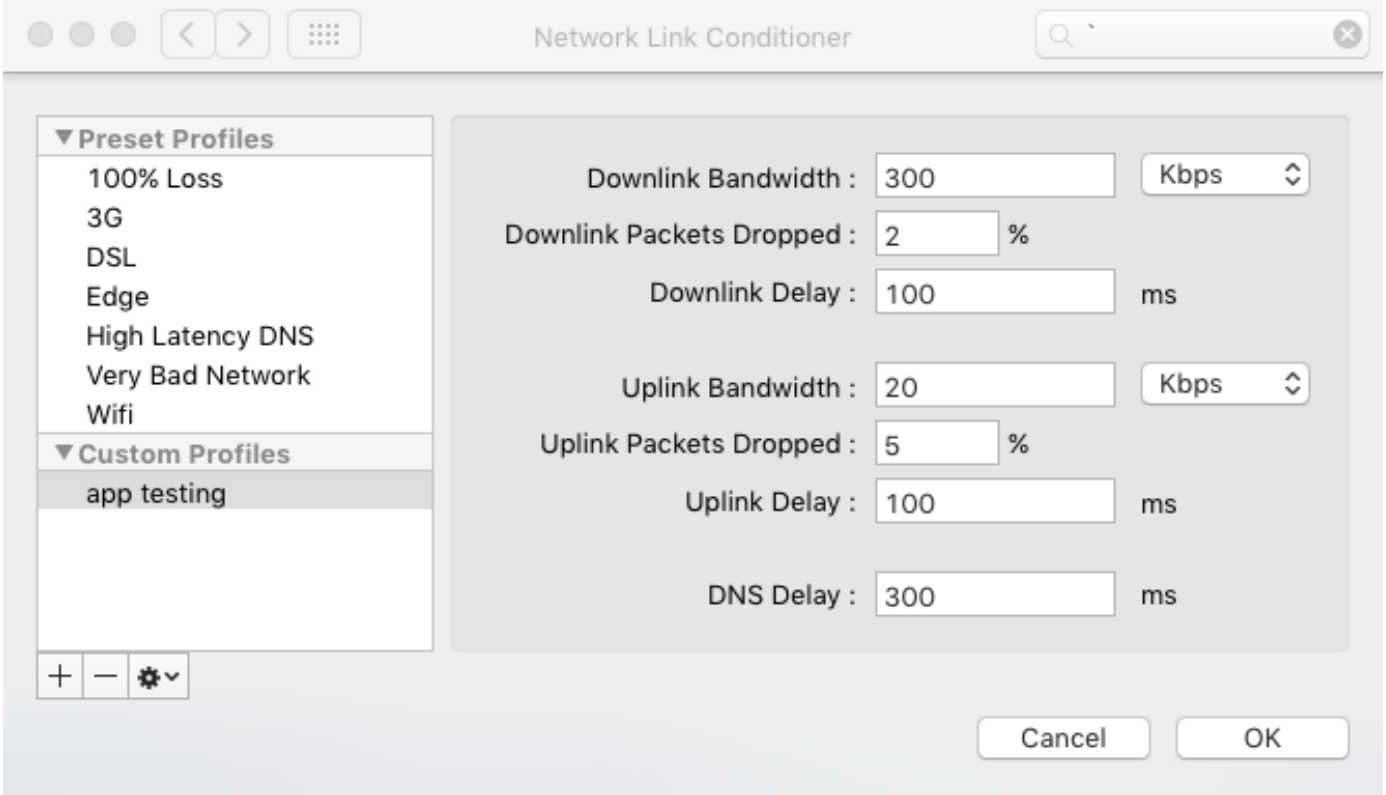
5 Apple Link Conditioner

The Apple link conditioner is a network impairment tool offered in the ***Hardware IO Tools for Xcode*** download, available from the Apple Developer Center. After installation, the app appears in the system preferences:

And opens with an impairment summary dialog that has an On/Off button:



It comes with standard Profiles, and new custom profiles can be added:



The link conditioner applies impairments to all packets entering or existing the network stack. The impairments apply to all traffic to/from browser-based apps, or stand-alone Mac apps.

Pros

- The impairment mechanism is conveniently built into the network stack.
- It's relatively easy to turn the impairment on/off using the button.

Cons

The impairment mechanism is missing several essential settings:

- Rate limiter queue drop
- Rate limiter WAN packet overhead
- Jitter
- Packet corruption
- Packet duplication
- Packet resequencing.

And several impairment capabilities are missing:

- Burst mode

- Waveform specification
- API scripting
- Transitional control
- Impairment accuracy

The impairment mechanism has no packet classification:

- The impairments apply to all packets traveling to/from the Mac.
- It's not possible to perform classification based on the VLAN tag.
- It's not possible to limit the impairments to one type of packet, to isolate bugs: all traffic gets impaired at once, including DUP / TCP / HTTP / DNS / DHCP / ARP.

6 Netem

Available on Linux machines only, Netem is a special queue discipline (qdisc) module that can be used within the Linux Traffic Control (tc) system. The Netem qdisc module provides basic impairment functionality, including bit-clocked rate limiting, but not token buffer rate limiting: To achieve a typical network impairment configuration, you would use the existing Linux TBF (token bucket filter) qdisc to impose token buffer rate limiting, then add the Netem qdisc to provide other basic network impairments. Netem can be used in one of two configurations:

Single-ended: If your application is running on Linux, then Netem can be configured to impair packets flowing to/from the application and an Ethernet interface. This setup is obviously limited to apps, IDEs, or device emulators running on Linux; as a result, this configuration is seldom used for app testing.

Double-ended: In this setup, a Linux machine is configured as a bridge, using two physical Ethernet interfaces, and the machine is used as a stand-alone appliance. Configuring Netem in this way requires Linux expertise:

- This setup requires two instances of Netem: one for each interface.
- A Linux bridge must be configured between the Netem instances
- The Linux iproute2 toolset must be used for packet classification.
- It is necessary to use the Linux Traffic Control (tc) system to wire everything together.

Pros

The Netem qdisc, plus the standard token buffer filter qdisc, provide some essential impairment settings, with stochastic burst features.

Netem can be configured on a stand-alone Linux machine operating as a bridge, to act as an impairment appliance for an app.

Cons

Netem has complex installation, configuration, and maintenance requirements:

- Netem is highly dependent on the Linux Kernel version. Newer Kernel updates can cause deviations in Netem performance and behavior. In addition, certain Kernel tuning parameters must be tweaked to get Netem to work well. The resolution of some Netem parameters, such as delay, may be limited to multiples of 10ms, unless the Kernel is tuned in a specific way.
- The performance of Netem is dependent on the underlying machine. Netem will start to drop packets if it runs out of CPU horsepower. If Netem is moved to a slower machine, it will start dropping packets sooner, and you may need to revise your test plan to

accommodate the change in performance. If you move Netem to a new machine, and you see your app dropping packets, it might not be your app; it might be Netem.

- The Linux bridge that links the two Netem instances has certain limitations; it may absorb/drop certain packets, without passing them. It also may generate new spanning-tree packets.

Using Netem is complicated:

- A mastery of the Linux iproute2 tools is required to classify and tag packets for Netem.
- The tc (traffic control) Linux command configures netem, and it is not intuitive. The command to set a simple rate limit is:
  tc qdisc add dev eth0 root netem rate 5kbit 20 100 5
- Netem provides no visual feedback. It can print statistics, but there is no graphical indication of what is going on inside each impairment. In particular, it's not easy to monitor the input queue level of the delay impairment or rate limiter impairment. For the rate limiter, as the input queue builds up in size, the effective delay imposed by the rate limiter increases, and you get both a rate limiter functionality and a delay functionality. With Netem, it's not easy to determine the effective delay in real time. If the queue size of either impairment is small, the impairment may begin dropping packets before you realize it. Without convenient feedback of what the input queue is doing, you will likely spend your time debugging the netem queue, not your app.
- Netem uses non-intuitive units: the units of "kbps" denotes 1024 bytes per second, not 1000 bits/sec.
- All of these complications are magnified when sequencing through a series of test scenarios that use different parameters.
- Implementing a rate limiter in Netem can be complex. In order to implement a rate limiter with RED (random early detection queuing), TBF (token bucket filter), and WAN overhead computations, you may need to string together three qdisc modules: the RED module, the TBF module, and the Netem module.

There are also issues with Netem performance, accuracy, and behavior:

- Each new version of the Linux Kernel includes a new version of Netem, and the behavior of Netem changes with each version. Newer versions allow reordering packets caused by jitter delay; older versions will retain the original order.
- Packet classification is limited to IP address only. There is no facility to classifying packets based on other IP fields or anything at layer 2.
- Linux Traffic control is limited to the resolution of the Linux timer. In the most limited case,

the default Linux timer is set to 100Hz, which allows the rate limiter to send packets at a rate no greater than 100 packets/sec. In contrast, for very high-bandwidth applications, it is essential to be able to test packet streams consisting of small packets, at the rate of 200,000 packets/second.

- It is critical for an impairment parameter to seamlessly switch from one value to another, without a glitch in the impairment processing. Otherwise, you will wind up debugging the glitches in the impairment emulator, not the bugs in your app. However, Netem guarantees no continuous, repeatable behavior when any parameter changes: changing parameters results in a brief "unknown" state during the switchover.

Before using Netem, it's necessary to validate your Netem installation, so that you know how accurate the impairments will be:

- Netem does not compensate for all delays that may affect a packet as it flows from one interface to another through the Linux machine. To impose accurate metrics on a Netem instance, it's necessary to profile actual delays through a stand-alone machine, then manually compensate for these deviations when issuing configuration commands via the command line. A delay configuration of 10ms may result in a total interface-to-interface delay of 20ms (in the worst case), and a manual adjustment is required to correct for the deviation.
- Fully characterizing Netem performance requires measuring the amount of variance that Netem exhibits with each metric. Setting a delay of 10ms may result in delays anywhere between 8ms and 12 ms. That addition random variation adds to the unintended jitter. Which means you might need to lower your impairment jitter parameter to compensate for additional jitter imposed by other impairments of netem.
- A packet may experience additional unknown delays, including additional jitter, as a result of internal buffers used by the network adapters, and by the Linux packet processing routines. If these buffers are changed / reconfigured, the packet delays and jitter may change, and it may be necessary to manually correct for them.

Some critical features are missing from Netem:

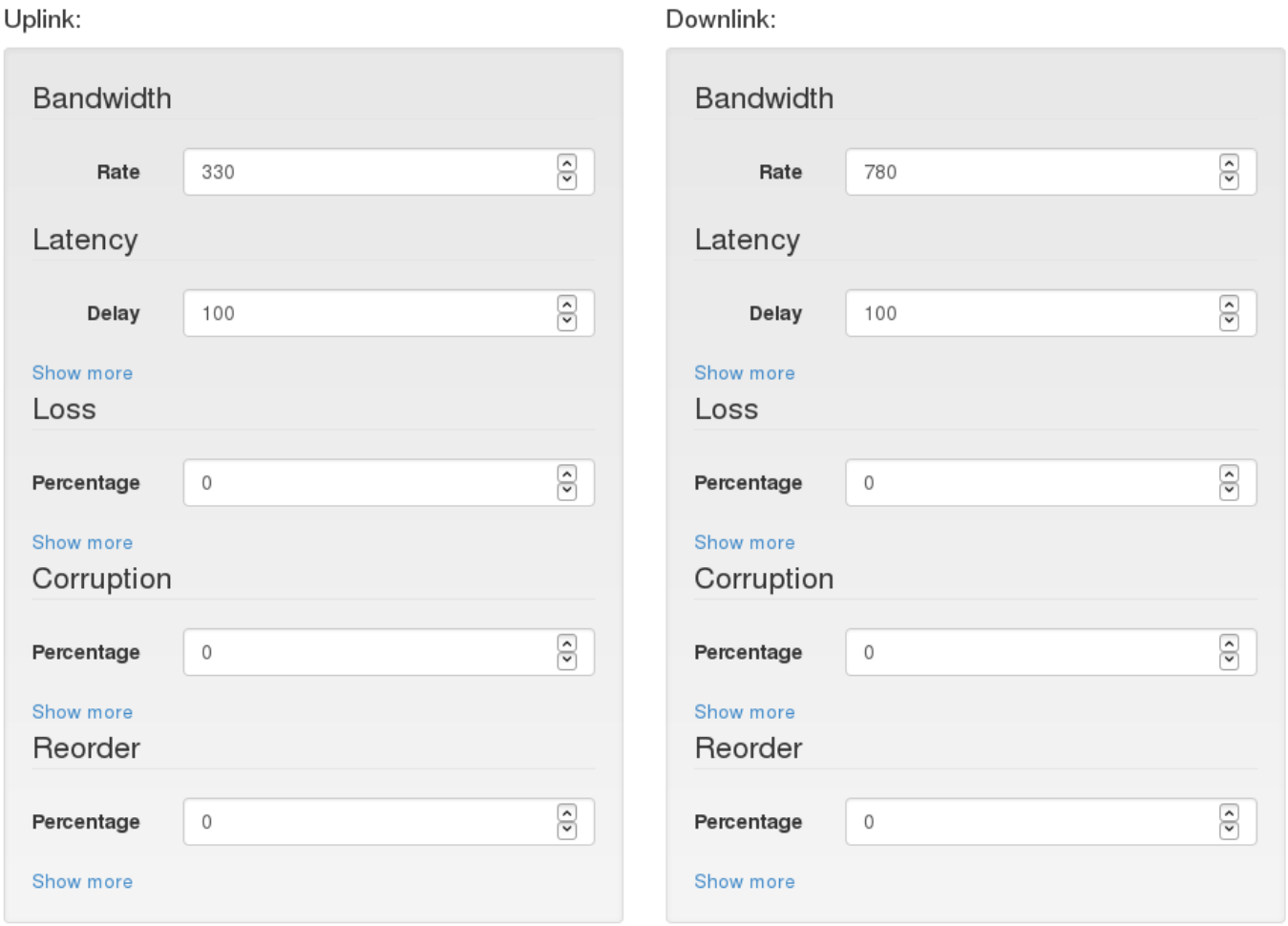
- Packet resequencer: Netem offers two simple re-ordering schemes: 1) For the delay impairment, a percentage value determines the number of packets that get no delay, and those packets will be forwarded before delayed packets. 2) It's possible to instruct Netem to swap the order of every Nth packet. However: One critical feature for app testing is a resequencer capability, which stores up a certain number of packets, then forwards the packets with a custom reordering. Netem does not have this feature.
- Transitional control: Disabling the Netem delay or rate limit functionality may cause packets in the input queue to be dropped, rather than correctly forwarded.
- Netem offers no waveform expression setting for parameters: time-varying impairments

must be implemented by writing code that issues Linux tc commands.

- Because Netem is a layer 3 gateway, it cannot be inserted into the network path like a layer 2 switch: The DUT endpoints must be configured to use Netem as a gateway.
- Because there is no GUI, there is no convenient A/B test bypass button: you can't easily disable Netem without issuing command line instructions.

7 Facebook Augmented Traffic Control (ATC)

The Facebook ATC is a customized layer 3 gateway that runs on Linux. One interface connects to the LAN / DUT, and the other connects to the WAN/server/internet. The gateway offers a set of impairments for each direction. ATC passes the packets through a process to perform bandwidth throttling, then sends the packets through Netem, an impairment emulator built into Linux. Many of the problems listed for Netem also apply to this approach. ATC uses the Linux iptables firewall to mark packets, based on IP address only (not the port number). The impairment mechanism applies different sets of impairments based on the marking. ATC comes with a web server to provide a browser GUI for a basic configuration that does not offer classification:



Clicking on the ***Show more*** links reveals the Netem coupling fields, to emulate a sequential burst of impairments. Adding packet classification requires manual configuration file editing.

There are two basic network setups for ATC:

- For wired clients, because the impairments run inside a layer 3 gateway, the DUT must be configured to use the ATC gateway to connect to the upstream server.
- For wireless clients, a wireless access point is typically used as the LAN interface for ATC.

Pros

The ATC is a stand-alone device that is separate and independent from the app, which makes testing easier.

The ATC provides some essential impairment settings, including packet drop, corruption, and reorder, with stochastic burst features.

It comes with a browser-based GUI, and easy access to an on/off button for A/B testing.

ATC offers a scripting API, either for automation, or for building a new GUI front-end.

Cons

ATC is an unsupported open source project, with major caveats:

- ATC has a discussion form, but there is no formal support.
- Installing ATC requires Linux expertise, and in some cases requires changes to the underlying code, in order for it to work on some Linux distributions.
- ATC uses Netem as the impairment engine, which has not been stable over time. Netem is seriously impacted by kernel version, network interface hardware, and platform type. Among other things, Netem requires special Kernel tuning, and this tuning is not incorporated into ATC. Unless the underlying platform is properly constructed and tuned, Netem can give either inaccurate results or go awry in various ways. In a testing environment, test engineers should not spend time hunting down unexpected behavior in the test tool. Any change to the kernel version, hardware platform, or network interface hardware would require re-validation of Netem, according to the items mentioned in the chapter on Netem.
- Updates to the Linux Kernel can cause problems with the operation of netem.

The ATC is missing several essential impairments:

- Rate limiter queue drop
- Rate limiter WAN packet overhead
- Jitter
- Packet duplication
- Packet resequencer. ATC offers simple re-ordering: the percentage value entered in the reorder field determines the number of packets that get no delay, and those packets will be forwarded before delayed packets. But ATC it does not have an input queue that can accumulate packets, then forward them with a new sequence order.

And several impairment capabilities are missing:

- Transitional control

- ATC offers no waveform expression setting for parameters: any time-varying impairments must be implemented by writing code to the API.
- Because the ATC is a layer 3 gateway, it cannot be inserted into the network path like a layer 2 switch: The DUT endpoints must be configured to use the ATC as a gateway.
- Since the impairment engine is software-based, impairment accuracy is limited, especially for higher bitrates.

Packet classification is limited in flexibility and usability:

- Adding packet classification and different impairment presets requires manual configuration beyond what the GUI provides.
- Packet classification is limited to IP address only. There is no facility to classifying packets based on other IP fields or anything at layer 2.

8 KMAX impairment engine

KMAX is a dedicated hardware platform that operates as an invisible layer 2 bridge, inserted between the endpoint and the server:

There are two basic network setups for KMAX:

- For wired clients, no additional configuration is needed: KMAX is inserted between an endpoint and the server in the network path.
- For wireless clients, a wireless access point is typically added in front of KMAX.

For classification, KMAX is able to classify packets based on any header field at any network layer, or based on payload data at specific payload offsets. It's possible to impair packets destined for certain IP addresses and ports, and to classify based on UDP or TCP protocol. It's also possible to classify packets based on VLAN tag.

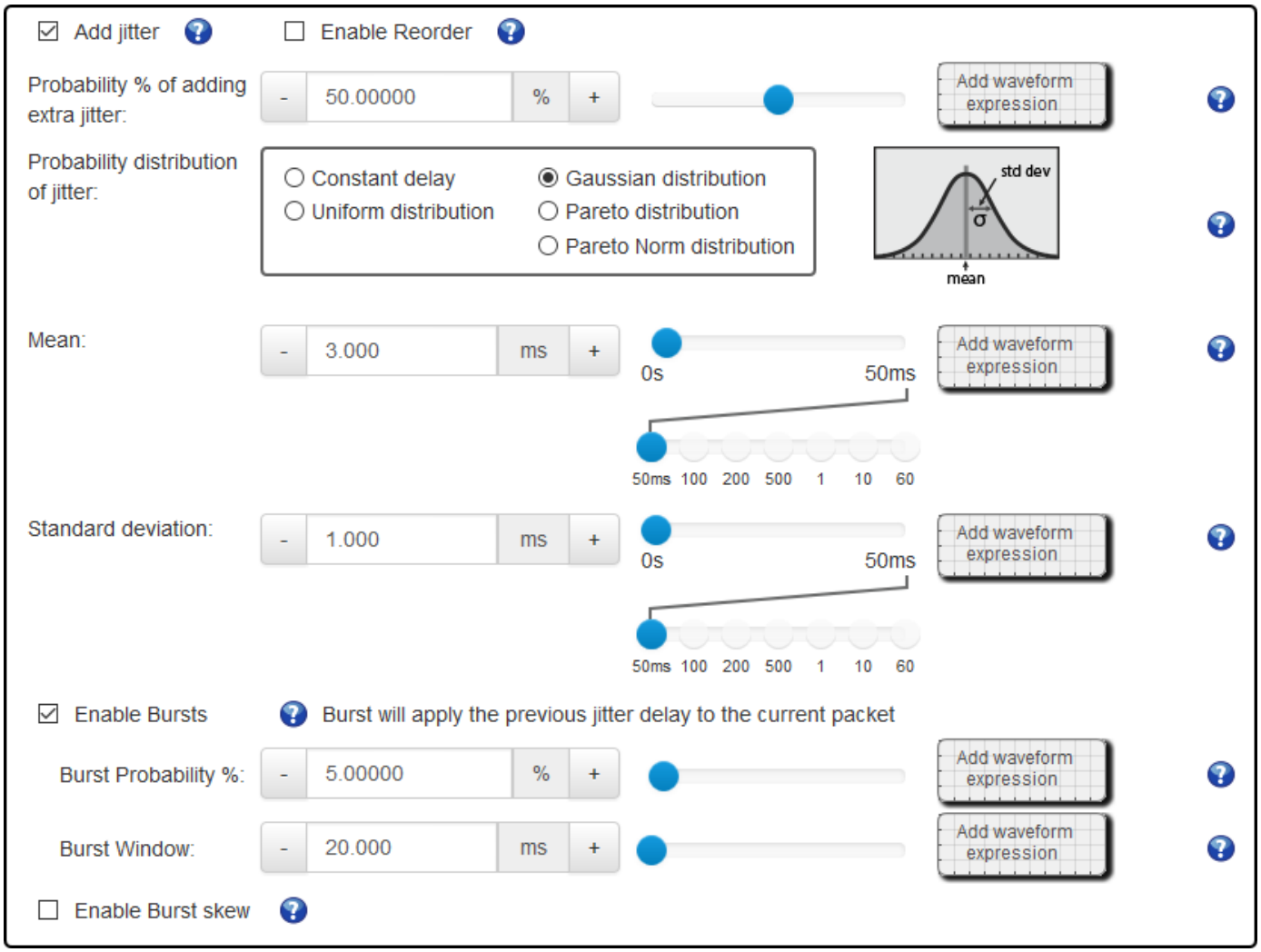
For impairments, KMAX offers all recommended impairments for app testing:

- delay
- jitter
- resequencing
- drop
- duplication
- corruption
- rate limiting, with WAN access link input queue drop, and WAN link overhead emulation, plus RED (Random early detection), token buffer, and bit clocking.

KMAX offers the full range of impairment capabilities:

- burst mode on any impairment
- waveform expressions for all parameters
- Asymmetric settings for all parameters
- Easy A/B testing, using buttons on a mouse hooked up to the unit
- API scripting
- Transitional control
- High impairment accuracy

For easy of use, KMAX provides a browser-based GUI interface. This is an example of the jitter settings dialog:



Pros

- Simple network setup: layer 2 invisible; no endpoint configuration necessary.
- Flexible packet classification: packets can be classified using any content at any layer.
- Includes all impairments for app testing, including advanced capabilities like WAN queue drop emulation and resequencing.
- Full range of impairment capabilities: asymmetric settings, waveform expressions, burst, A/B testing
- Comprehensive browser GUI, with no need for manual configuration files.
- API scripting support for integration into a test harness

Cons

- Not free or open source: KMAX is a dedicated hardware product.

9 Capability matrix

Each solution occupies very different impairment testing categories:

- Chrome DevTools: low complexity, very limited feature set, limited to in-browser apps.
- Netem: Extremely high installation / tuning / maintenance overhead, medium-high feature set for Web apps, PC/Mac apps, or mobile apps.
- ATC: high complexity, high maintenance, medium feature set for Web apps, PC/Mac apps, or mobile apps.
- Apple Network Link Conditioner: low complexity, limited feature set, limited to Web or Mac apps.
- KMAX: higher cost, greater ease of use and full feature set for Web apps, PC/Mac apps, or mobile apps.

See the next page for a matrix summary:

The feature matrix highlights the specifics of each solution:

Impairments

	Chrome DevTools	Facebook ATC	Netem	Apple Link Conditioner	InterWorking Labs KMAX
Rate limit: up/down BW	✓	✓	✓	✓	✓
Rate limit: input queue packet drop			✓		✓
Rate limit: header emulation			✓		✓
Delay	✓	✓	✓	✓	✓
Drop		✓	✓	✓	✓
Jitter			✓		✓
Corruption		✓	✓		✓
Duplication			✓		✓
Full Resequencing					✓

Capabilities

	Chrome DevTools	Facebook ATC	Netem	Apple Link Conditioner	InterWorking Labs KMAX
Ease of A/B testing	✓	✓		✓	✓
Ease of use/installation	✓			✓	✓
All parameters asymmetric		✓	✓	✓	✓
Scripting		✓	✓		✓
Burst mode		✓	✓		
Waveform expressions					✓
Transitional control					✓
High accuracy					✓
Flexible Packet classification					✓