ECN-Bits Reader Library (dev. by Thorsten Glaser)

It can be found here: `https://github.com/tarent/ECN-Bits` (see Download below)

About

The ECN library is a collection of utility functions and examples that demonstrate, for various operating environments, how to get the ECN bits (which are part of the IP traffic class) out of the IPv6 or Legacy IPv4 header of a datagram (UDP) packet.

Development of the ECN library was funded by Deutsche Telekom. It is published as Open Source (see below) under Free licences, so it can easily be embedded into applications of any kind.

For TCP connections, the operating system already takes care of signalling back ECN information using the ECE (ECN echo) and CWR (congestion window reduced) mechanisms native to ECN with TCP. Therefore, this library is not useful for, or even usable on, TCP sockets.

Use case

This functionality would be used by adaptive managed latency-aware applications, such as video streaming, as follows: The client application, running in user equipment (such as a laptop, multimedia box, or smartphone), would collect the percentage of received packets that have the ECN “CE” (congestion experienced) bit combination set, and report that regularly (e.g. every 20–50 ms) to the server, which can then use a rate-adjusting algorithm (such as SCReAM) to reduce (if congestion was experienced) or increase (up to the connection maximum) the data rate (e.g. video stream quality and pixel size). Similarly, the server can utilise this technology to verify that the packets sent from the client back to the server are not congested.

Alternatives

On operating systems that do not support retrieving the IP traffic class, using a “raw” socket for receiving datagrams and decoding the IP header manually is discussed at several places in the internet. This is a rather fragile method.

For most operating systems, a packet sniffer can be used to retrieve the information from the IP header of a connection as well. This method has multiple concerns (security-wise, a packet sniffer is always problematic; added latency from the sniffing process; a disconnect between retrieval of the packet payload and other metadata vs. the sniffed IP header) but is a valid alternative for testing. This is persued independently in another project. (Which will be linked to from here once available.)

Technical details

Caveats

Operating system-specific

Microsoft® Windows®

Windows does not support setting the traffic class on *outgoing* packets **at all**. This also affects WSL 1 (Windows Subsystem for Linux, i.e. running Linux applications unmodified on Windows, a.k.a. “bash for Windows”). Retrieving the traffic class is possible in all supported versions of the operating system; it has been tested on Windows 10.

WSL 2 is a Linux virtual machine with separate networking; if one can get it to network at all (lack of IPv6 support, lack of support for UDP portforwarding, broken default DNS resolver), it works like “proper” Linux, i.e. can set the outgoing traffic class on packets.

NetBSD®, OpenBSD and derivatives thereof

These operating systems do not support receiving the IP traffic class. **FreeBSD®** and derivatives thereof, such as **MidnightBSD**, however, *do* support this. The library has been tested on MidnightBSD 1.2 and 2.0-CURRENT so far.

Solaris

These operating systems reportedly support receiving the IP traffic class at least for IPv6, perhaps also for Legacy IPv4, so a port might be welcome.

Use of “v4-mapped” IPv6 addresses

Background: with “v4-mapped” addresses, an application can use an IPv6 socket to listen to not only IPv6 but also Legacy IPv4 connections. This can make application design easier (handling only one socket in a blocking way, instead of using `select` or `poll` on multiple sockets) but is inherently fragile and not recommended. The use case of retrieving the IP traffic class is especially tricky, as the system has to deal with Legacy IPv4 operations on presumably IPv6 sockets, which is not universally implemented.

FreeBSD® and derivatives disable the use of v4-mapped addresses by default, so this is not usually a concern on these operating systems. If enabled, however, the traffic class cannot be read from such connections.

Linux usually supports this use case on v4-mapped addresses. In fact, **Android** uses only IPv6 sockets internally and so relies on this. It is not, however, guaranteed to work on all Linux systems, so its use is nevertheless discouraged.

Windows does not support reading the traffic class from v4-mapped IPv6 sockets under Winsock2 (the native sockets API). On the other hand, applications running under **WSL 1** *do* support this, mirroring Linux support. (**WSL 2** is unmodified Linux, i.e. supports this as well.)

Darwin (**Mac OSX**) also supports v4-mapped sockets, fully, except that per-socket setting (`ecnbits_tc`) of DSCP bits is not possible, only the ECN bits of the traffic class are set; using `sendmsg(2)` with `ecnbits_mkcmsg` to set them per packet, however, works (see `c/client`).

Supported operating environments

Please refer to the associated `README` files in each solution for further information.

C / Command line: Unix (Linux including WSL, Darwin/Mac OSX, FreeBSD/MidnightBSD, Winsock2 on Windows)

The solution for unixoid (BSD sockets) or Winsock2 environments is comprised of a library (which can be included in applications) and two example programs, a client and a server.

This solution comes in two flavours: the standard Unix solution (for use with BSD sockets) and one adapted for Winsock2. The latter *does* function on all systems supported by the former, but it includes massive amounts of compatibility definitions and glue code to make things work under Winsock2 as well, so it’s too complex to be a good example for regular Unix systems. In addition, the Winsock2 variant omits the functionality needed to support setting the outgoing IP traffic class, as this is unsupported under Windows and can lead to failures to send packets if attempted.

The standard solution is shipped in the `c/` subdirectory of the repository, the Winsock2 variant in the `ws2/` subdirectory.

The library

… consists of a C header file (in the `inc/` subdirectory) and a static and, for Unix, shared library, as well as associated documentation in the form of manual pages, as is customary for Unix environments. The library source code and manual pages are located in the `lib/` subdirectory.

This library can be used from all programming languages and frameworks that allow calling C libraries, either directly (e.g. from C++) or via a foreign function interface (e.g. from Python3). It contains functions for:

- setting up a socket (file descriptor on Unix, `SOCKET` handle on Winsock2) for use with the library
- (Unix only) setting the default outgoing traffic class on a socket (but check above regarding v4-mapped sockets)
- a native function to receive data from a socket and write the traffic class to a supplied extra parameter, which functions like `recvmsg()` but fills in the `msg_control` and `msg_controllen` members of `struct msghdr` (Winsock2: the `Control` member of `WSAMSG`) itself if necessary
- wrappers around the functions `recv()`, `recvfrom()` and `recvmsg()` as well as `read()` as applied to sockets (with a small semantic difference regarding zero-length datagrams) that also write the traffic class to a supplied extra parameter
- macros to validate the received traffic class and split it into DSCP and ECN bits, as well as formatting them into human-readable texts
- utility functions:
    - (Unix only) prepare a control message for use with `sendmsg()` to set the outgoing traffic class for one specific packet
    - (Winsock2 only) replacements for the functions `sendmsg()` and `recvmsg()` using Unix-style semantics
    - (both) determine whether a socket is IPv6, Legacy IPv4, or something else

To use the library within a framework, the framework must expose the bare file descriptor (Winsock2: `SOCKET` handle) so the library call to prepare it can be used, and the framework’s method of receiving packets must be replaced with one of the wrapped calls. (The Android library (see below) initially did something like this, but it eventually had to reimplement the entire network I/O stack creating its own socket file descriptors.)

The example server

The server is called with just one option (the port number to listen on on all interfaces) or two (the IP address and port number to listen on). It then runs in the foreground, waiting for incoming packets until it is aborted with `^C` (Ctrl-C). The nn-Winsock server also accepts an argument for what DSCP bits it should set in response packets.

When a packet arrives, it displays one line of information about the packet (see below) and sends **four** (Unix) packets or **one** (Winsock2) packet back. The return packets contain most of the information displayed (see below) *except* for the user data. The four packets sent back by the Unix server differ only in ECN bits (00, 01, 10, 11 are sent in this order); the Winsock2 version does not set the outgoing ECN bits for the reasons outlined above.

Besides the example client, suitable packets can be sent with netcat-openbsd.

The information displayed (and sent back) is:

- the timestamp of when the server received the packet, in server-local time using the UTC timezone
- whether the packet was truncated (if the internal buffer was too small) for either user data or control data (traffic class); this should always read `no trunc` because the buffers should be sized large enough and is useful for debugging
- the source address of the packet (the IP address in square brackets, followed by a colon and the port number)
- which ECN bits were set on the received packets as follows, plus the IP traffic class octet as two-digit hexadecimal number in curly braces (or double question marks if it could not be determined)
    - `no ECN`—00
    - `ECT(0)`—10
    - `ECT(1)`—01
    - `ECN CE`—11
    - `??ECN?`—unknown, could not be determined from the packet
- the user data, i.e. the payload of the packet received, without a trailing newline, between angle brackets (this is not sent back)

The example client

The client is run with two arguments: the IP address (or hostname) of the server to connect to, and a port number. The standard variant also accepts an optional third parameter setting the outgoing traffic class.

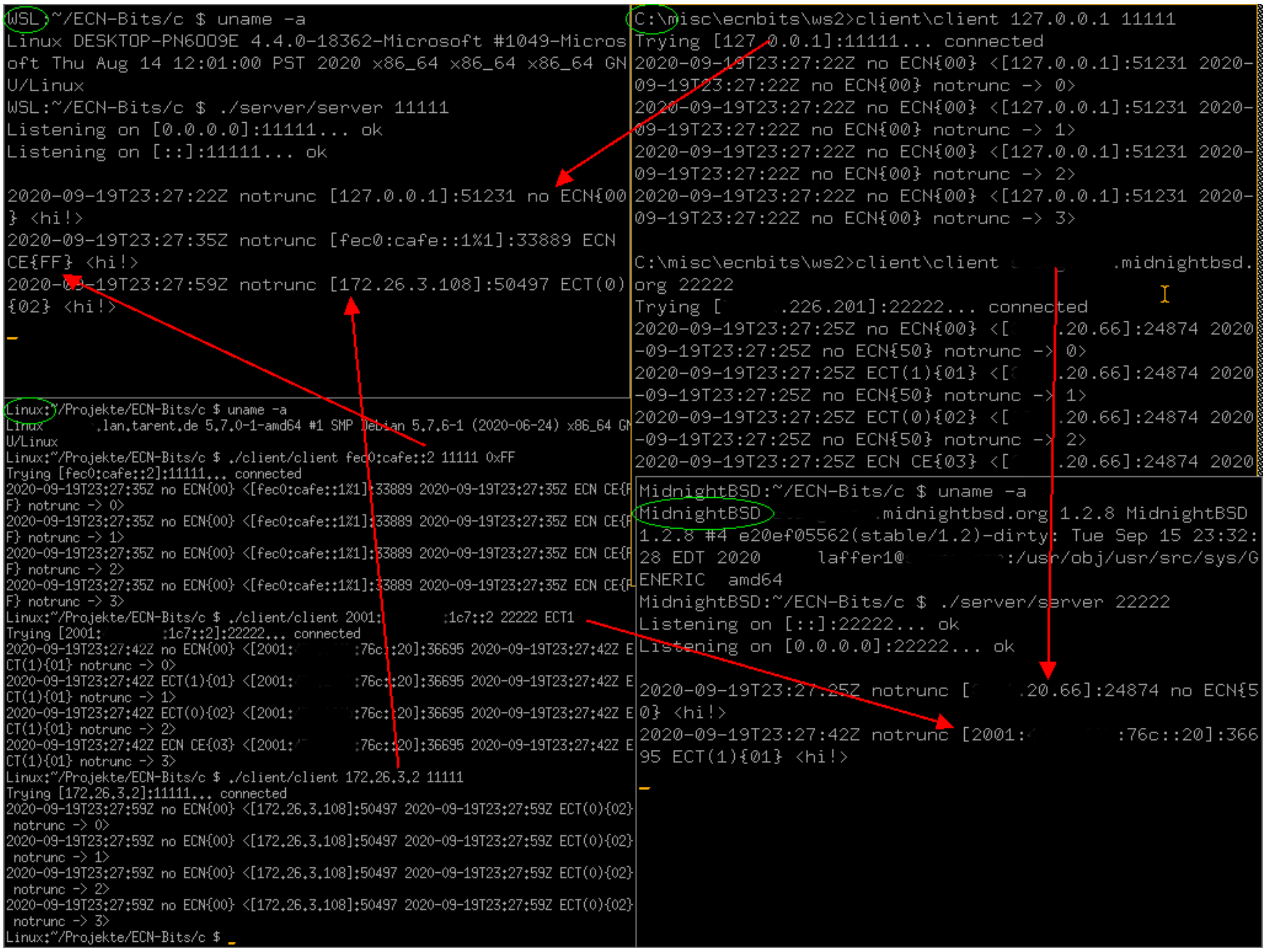
The client then sends one packet with the payload “hi!” to the server (if the server resolves to multiple addresses, all are tried in order) and waits for incoming packets from it. One second after the last packet was received, it tries the next address or terminates. If no address could be reached, it issues an appropriate error message.

The client also displays, for each packet, a line consisting of:

- the timestamp of when the client received the answer packet, in client-local time using the UTC timezone
- which ECN bits were set and the traffic class octet in hex (see above)
- the payload between angle brackets

The client can also be tested against, for example, the `daytime` (udp/13) server built into `inetd`, and netcat-openbsd (in listening mode).

Screenshot



The screenshot shows four terminal windows:

- upper left: Unix C/CLI server compiled as Linux application, running in WSL 1 on Windows 10
- upper right: Winsock2 C/CLI client compiled with Visual Studio, running on Windows 10
- lower left: Unix C/CLI client running on Debian GNU/Linux
- lower right: Unix C/CLI server running on MidnightBSD

The red arrows show which client invocation led to which server display line.

Legal

The entire C/CLI solution, in either variant, is published under The MirOS Licence and thus Open Source using the permissive model (use only requires attribution).

Java & JNI—`DatagramSocket`, `DatagramChannel`: Android

This solution for Android is comprised of two modules (the AAR library and an APK demo äpp) in a standard IntelliJ (Android Studio) project and is shipped in the `android/` subdirectory. It is designed for Android 8 “Oreo” (the oldest currently supported release) and newer, and is known working on Android 9 and 10 as well, possibly working on Android 11 as of 23 December 2020.

The AAR library is not published to a central Maven repository but designed as a component that can be embedded into other projects (with a few clicks in Android Studio). It comes in the `ecn-lib/` subdirectory, whereas the `app/` subdirectory holds the example GUI application.

Technical difficulties

We could identify three ways Android developers would communicate using UDP (feel free to suggest more):

- `DatagramSocket`
- `DatagramChannel`
- Netty

We also expect use of libraries, such as WebRTC libraries utilising RTP over UDP. To what amount such libraries can be patched to call the ECN-Bits library functions remains to be seen.

The first approach at integrating ECN-Bits support into the Java™ frameworks was minimal-intrusive: subclass the existing implementations and change only what's necessary. Unfortunately, both OpenJDK and Android threw wrenches into this work: OpenJDK does not allow placing classes into the `java.base` Jigsaw module, which `java.net.*` belong to, and Google are hiding functionality from Android SDK users (that is, regular äpps), permitting access only for the Android runtime environment itself. This was successful for `DatagramSocket` only and unfortunately affected every `DatagramSocket` created by the äpp, not only those that were explicitly created as `ECNBitsDatagramSocket`.

The second approach involves implementing the complete lifecycle of a datagram channel and socket. This is much larger in scope (although we already explicitly excluded IP Multicast) and necessarily cannot mirror completely what the system does (again due to lack of access e.g. to Android's `CloseGuard`) but comes very close. (Thankfully, the implementations changed very little between Android 8 and 11, allowing us to use the Android 11 code as a base.) This approach allows the use of `DatagramChannel` including its `socket()` adapter, as well as *almost* drop-in compatible support for `DatagramSocket`; this should allow both OIO and NIO modes of Netty to work as well.

This approach is currently being finalised and will also be the base of a desktop/server JRE solution. Unfortunately, some methods are still not called or callable, so it's only close to being a drop-in solution.

The best fix would be a Google-provided API to access the IP traffic class, ideally even in stock OpenJDK. No such API exists as of the time of this writing.

Network difficulties

Note that networks may not pass along the ECN bits (or the IP traffic class byte really). In particular, the **Android emulator** does not support passing that information along. For testing, you can run the client äpp against a CLI server running under `adb shell`; it would be best to test this on bare metal (real physical smartphones); note that most phones will refuse the installation of an unsigned äpp, so either install the debug APK or `apksign` the äpp before installing.

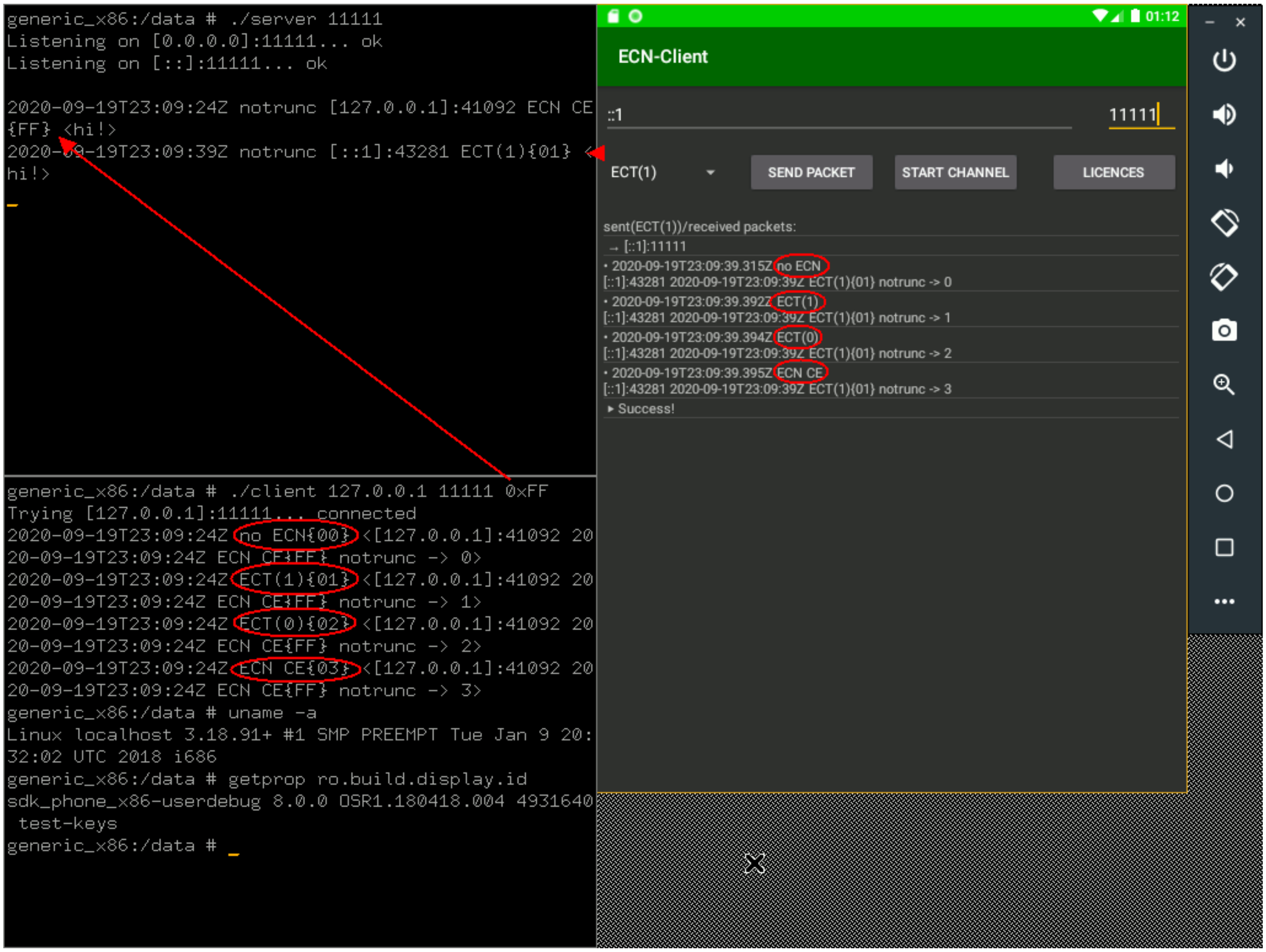
The example application

The example Android äpp is only a client. It draws a UI which has input fields for IP address or hostname, port, and outgoing traffic class, like the C/CLI example client. It also has two buttons:

- Send Packet: uses `ECNBitsDatagramSocket` to send out a packet and wait 1 s for answers, like the CLI client does
- Start Channel: uses `ECNBitsDatagramChannel` to continously send and receive packets in multiple threads
- Licences: displays what Google's standard library thinks are the licences of the dependencies of the äpp

Below the buttons, textual output (output lines separated with thin lines) will be shown; the content is similar to what the C/CLI example client shows, except the payload is displayed after a linebreak (portrait format) or ‘’ separator (landscape format).

Screenshot



This screenshot shows an AVD emulator on the right-hand side, which runs the example client äpp. On the left-hand side, it shows two `adb shell` windows, the upper executing the C/CLI server, the lower executing the client. The first server line was caused by the CLI client, the second by the Android äpp client.

The C/CLI client and server used in this example were compiled to match the target CPU (x86 for the emulator, ARM for real hardware) and **statically** linked using musl libc, so they run directly on the Linux kernel that underlies Android, without going through the Android frameworks.

Legal

Most of the Android solution, as much as copyright law applies to it (some parts are from the Android Studio template or SDK documentation), is, like the C/CLI solution, published under The MirOS Licence and thus Open Source using the permissive model (use only requires attribution).

Some parts of the Android library are copied from the system implementation. These parts are published under the GNU GPL, version 2 only, with the GNU Classpath Exception, as outlined in its `LICENCE` file. As long as you only use the library in an otherwise independent project, the conditions of the GPL do not apply to the whole, but you will have to give the source code of the ECN-Bits Android library to recipients of your program. (If you don't modify the library, pointing them to the ECN-Bits project or bundling it alongside your software will usually be sufficient.)

The Android solution also includes parts of an independent module (Android libnativehelper) under the Apache 2 licence.

Note that *all three* of these licences apply to the ECN-Bits Android library, so *all three* must be adhered to: the Apache 2 licence for the "nh" module *only*; both MirBSD and GPLv2 with Classpath Exception for the OpenJDK/Android-libcore-derived code in this project, and just The MirOS Licence for the rest of this project.

Java & JNI—`DatagramSocket`, `DatagramChannel`: OpenJDK/JRE

Support for desktop and server Java™ has been implemented based on Android support. This is limited to some platforms, namely Linux, with the requirement to compile the JNI part for the target platform (as there is no "nice" way to provide JNI libraries for "all" possible platforms; using JNA in a second JDK solution is being discussed).

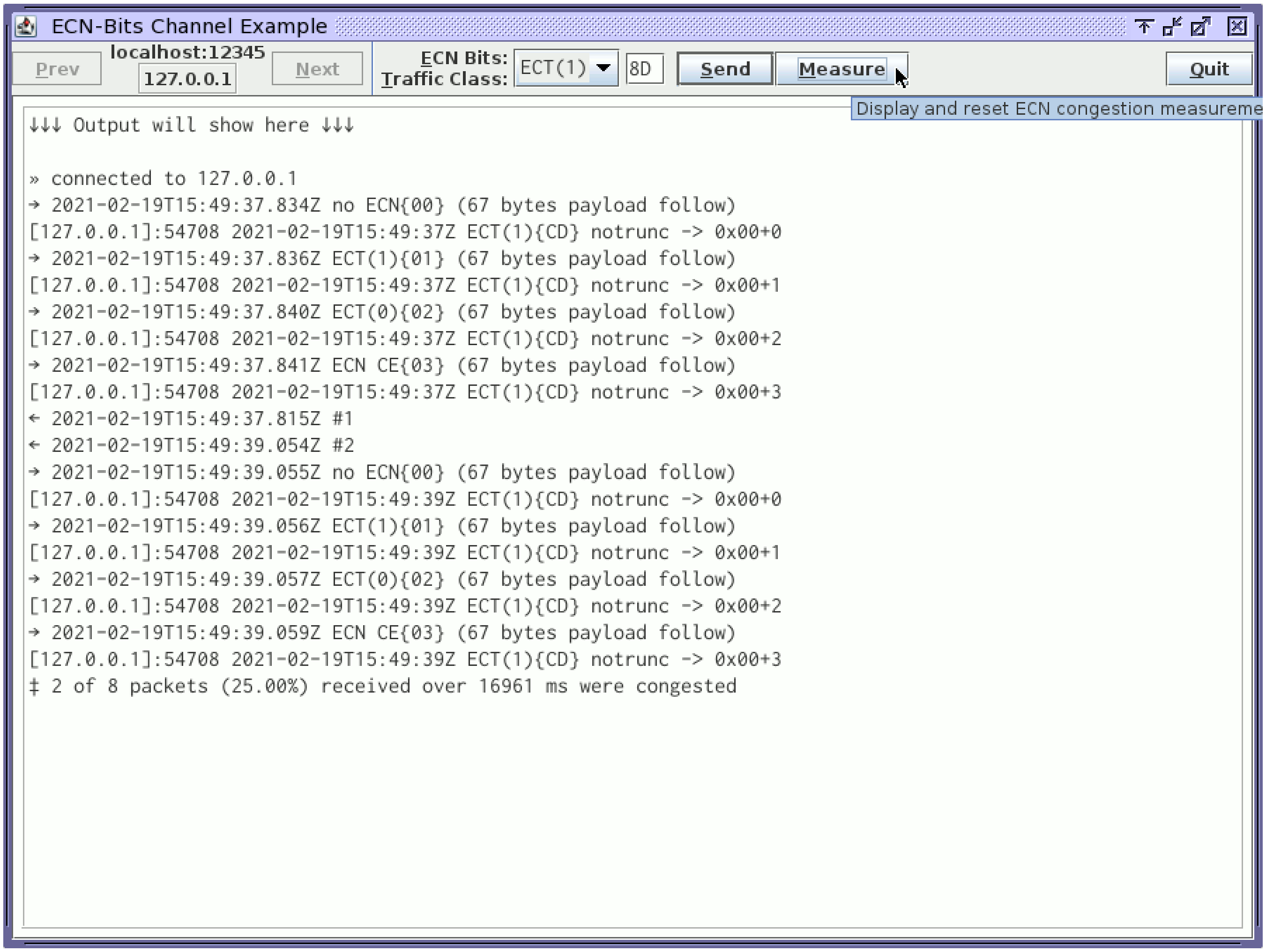
The example applications

The jdk-linux solution ships with two examples, again just clients, which are provided as `Main` classes, with small wrapper shell scripts to call them easily. See jdk-linux/README for more details on the options and arguments they take.

The DatagramSocket client (`./client.sh`) behaves in almost exactly the same way as the C/CLI example client does.

The DatagramChannel client (`./channel.sh`) is a Swing GUI application that can send out individual packets on button press, displays all incoming packets similar to how the Android äpp does it, can change the IP traffic class octet for outgoing packets on the fly and switch between the peer's IP addresses; it also shows congestion measurements.

Screenshot



This screenshot shows the `DatagramChannel` client against a C/CLI server running on `localhost` after having sent two and received eight packets and manually triggering measurement.

Legal

This solution is covered by the same terms as the Android solution.

C++ (and C): UWP (Universal Windows Platform)

Support is tentatively being implemented using the C library functions inside the WebRTC library.

C/VB.net: Windows/Mono .net Common Language Runtime

While support for this platform might be worthwhile, there currently is none yet.

Apple iOS

While support for this platform is worthwhile, there currently is none yet. Since this works well on Darwin, it will probably work out-of-the-box on iOS.

Utilising the library

See the individual solutions' `README` files for compilation instructions.

C solution

For environments that support inclusion of C libraries directly, such as C++, include the appropriate header `<ecn-bits.h>` (Unix) or `<ecn-bitw.h>` (Winsock2) after any necessary network headers such as `<sys/socket.h>` and network headers (Unix) or `<winsock2.h>` *and* `<ws2tcpip.h>` (Windows). Then link the application against `-lecn-bits` (Unix), `-lecn-bitw` (Winsock2 on Unix) or `ecn-bitw.lib` (Windows). Mind that the Unix library is available as static and shared library; use the static library (possibly build the library with `NOPIC=Yes`) if it is to not become a run-time dependency (or use `LDSTATIC=-static` to force linking the executables statically). The Winsock2 version deliberately does not build a shared library on Unix but it *does* create a Win32 DLL (link against `ecn-bitw_imp.lib` import library).

For environments with a foreign function interface, convert the information found in the appropriate header (see above) to FFI bindings and link against the appropriate library (see above) or dynamically load it.

At a minimum, call `ecnbits_prep()` to set up the socket and use `ecnbits_rdmsg()` or one of the wrapper functions to receive packets.

Please read the example client and/or server program source code to see how this works.

Darwin (Mac OSX, iOS)

Darwin Kernel Version 19.6.0 (Mac OS X 10.15.6 Catalina) is tested. Older versions might work. Newer versions might work… unless the library/loader-related changes broke anything relevant.

It is unknown whether this works on iOS, as it's not not intended for users to obtain local shell access, build and run CLI programs.

Linux 2.6.14+, FreeBSD 5.2+ (and derivatives like MidnightBSD), Windows 10

should work out of the box

Windows Server 2016 and newer

should work, untested

Solaris

can probably be ported to this system if there is interest

Android solution

Import the ecn-lib/ subdirectory into your project. According to the documentation it will become part of your project's source code.

Create an `ECNBitsDatagramChannel` or `ECNBitsDatagramSocket` instead of a simple `DatagramChannel` or `DatagramSocket` and use the return value of its `retrieveLastTrafficClass()` method with the `de.telekom.llcto.ecn_bits.android.lib.Bits` enum's static methods. To automatically collect statistics over received packets, the `startMeasurement()` and `getMeasurement(boolean continue)` methods exist.

`DatagramPacket` caveat

With `DatagramPacket`-using APIs (either an `ECNBitsDatagramChannel.socket()` or an `ECNBitsDatagramSocket`), if you wish to reuse the `DatagramPacket` you **must** (re)set the length before each call as no access to only the "received length" is provided to applications. This is true in general, both for OpenJDK and Android, but at least the latter lets you get away with not doing so for the stock `DatagramSocket` (although not for `DatagramChannel.socket()`, interestingly enough).

Missing functionality (as far as known)

- IP Multicast and all related socket options
- extended socket options
- Android `BlockGuard` and `CloseGuard`
- Android socket tagging, to some amount (`DatagramChannel`-style tagging is attempted)
- missing from `DatagramSocket` only
  - signalling of blocked threads
  - some retry handling
  - `peek()`—using `peekData()` instead
  - proper `dataAvailable()`—the overridden method would never be called, so it always returns `0`
- missing from `DatagramSocket` and adapted `DatagramChannel.socket()` only
  - setting (only) the "received length" in DatagramPacket (see above)
- some functionality is untested (binding, unconnected operation, disconnecting, peeking, most socket options, unblocked I/O, and exots like TTL, scatter/gather I/O)

Android 8, 9, 10, 11

… should work out of the box

OpenJDK/JNI solution

Compile the solution with “`mvn clean install`” then add an appropriate dependency to your project, for example this for Maven:

```
<dependency>
        <groupId>de.telekom.llcto.ecn_bits.jdk</groupId>
        <artifactId>jni</artifactId>
        <version>0.2-SNAPSHOT</version>
</dependency>
```

Adjust the version to match the one in the `jdk-linux/pom.xml` file. It is recommended to use a release version, not a snapshot version; however, the ECN-Bits repository is likely to always contain the latter due to technical limitations, but you can release it locally.

For gradle, this would be:

```
implementation 'de.telekom.llcto.ecn_bits.jdk:jni:0.2-SNAPSHOT'
```

It is also possible to add the solution as module to one’s own multi-module Maven project.

Everything else is identical to the Android solution.

This has been tested with OpenJDK 11 but should run on JRE 8 and above.

Download

The ECN-Library source code repository, which contains all of the solutions outlined above, including extra documentation, can be downloaded from the public GitHub project:

- `https://github.com/tarent/ECN-Bits`

Due to the fact that the libraries themselves are rather small, and the extra usage examples are the primary value of the repository, no pre-compiled binaries are distributed.