Evaluation of a libutp-based Tor Datagram Implementation

Karsten Loesing, Steven J. Murdoch, and Rob Jansen
karsten@torproject.org, steven.murdoch@cl.cam.ac.uk, rob.g.jansen@nrl.navy.mil

Tor Tech Report 2013-10-001
October 30, 2013

1 Introduction

Datagram designs are a promising approach to overcome Tor's performance-related problems. Advantages of datagram designs over stream designs are that they allow better end-to-end congestion management, reduce queue lengths on nodes, and prevent cell loss on one circuit delaying cells on other circuits. In earlier reports we compared possible Tor datagram designs [1] and outlined a testing plan [2].

In this report we evaluate our implementation of a libutp-based Tor datagram design. However, this evaluation does not focus on performance improvements over the current stream-based Tor implementation, because we found that our datagram-based Tor implementation is not mature enough for such a comparison. The focus is rather on our approach to integrate a datagram design into the Tor source code and on setting up test environments to evaluate the new design. We hope that our findings will be useful when further tweaking our libutp-based design or implementing other datagram designs.

In the next section we describe our libutp-based Tor implementation in sufficient detail for others to understand our source code and to act as blueprint for implementing other datagram designs. Sections 3 to 5 outline the experimental setup that we used to evaluate our implementation and point out roadblocks that kept us busy longer than necessary. Section 6 concludes the report by sketching out possible next steps.

2 Overview of our libutp-based implementation

We implemented a Tor datagram design using a slightly modified variant of Bittorrent's libutp library. libutp provides µTP connections which are similar to TCP connections but which are transported via UDP using the µTP protocol. In this implementation we don't fully replace TLS connections with µTP connections. Instead, we open a new µTP connection accompanying each TLS connection. The TLS connection is used for the handshake between client and

relay or between two relays, namely for VERSIONS, NETINFO, CERTS, AUTH_CHALLENGE, and AUTHENTICATE cells. All subsequent cells are then sent over the µTP connection.

libutp itself does not talk to the network, but defines interfaces for sending and receiving UDP packets. We implemented all network communication using libevent in four functions:[1]

- retry_utp_listener, called by retry_all_listeners, binds a UDP socket to the same port that we're using as OR listener port for incoming TCP connections.
- utp_read_callback is called by libevent to inform us of UDP packets received on the UDP socket which we immediately hand over to libutp.
- tor_UTPSendToProc is registered with libutp to tell us whenever it wants to send UDP packets to the network; this function does not send UDP packets directly, but buffers them and asks libevent to inform us when the UDP socket becomes writable.
- utp_write_callback is called by libevent to send out previously buffered UDP packets until the buffer is empty or the UDP socket is not writable anymore.

libutp provides a connection abstraction that is similar to TCP connections. The main concept is the µTP connection that we can open, close, read bytes from, and write bytes to. Opening and closing µTP connections happens in the following functions:

- channel_tls_connect opens a new µTP connection to a remote relay, possibly after deciding whether the remote relay is expected to speak µTP with us or not.
- tor_UTPGotIncomingConnection is called whenever a new µTP connection is opened by a remote client or relay.
- channel_tls_close_method closes a µTP connection when its corresponding TCP connection is closed.[2]
- channel_tls_free_method unsets callback functions on a µTP connection when the TLS connection is freed.[3]

Once a µTP connection is established, we can read or write bytes from/to it. These bytes do not include µTP headers which libutp adds and removes transparently for us. We wrote six callback functions that are called by libutp to read from and write to µTP connections and to inform us about about any state change of established µTP connections:

- tor_UTPGetRBSize is called when new bytes have been received on an existing µTP connection and libutp wants to find out if there's still space in our read buffer.
- tor_UTPOnReadProc is called to hand over newly received bytes.

[1] Are we missing a function that unbinds our UDP socket when Tor terminates? Should we do this in channel_tls_free_all?

[2] Does this mean incoming µTP connections that are never assigned to a TLS connection are never closed?

[3] Should free the µTP connection, too, unless libutp does that for us; and is this really necessary, or should libutp not call any callbacks anymore after we ask it to close a connection?

- tor_UTPOnWriteProc is called when libutp wants to write bytes to an existing μTP connection; note the additional level of indirection here: we're going to handle the part where we tell libutp that we want to write bytes to a μTP connection further down below.

- tor_UTPOnStateChangeProc is called when a μTP connection becomes writable[4] or when it becomes non-writable; if a μTP connection is destroyed, we also close the TLS connection associated with it.

- tor_UTPOnErrorProc informs us of errors with a μTP connection; we currently don't do anything with this information.

- tor_UTPOnOverheadProc informs us of the overhead produced by UDP headers, retransmissions, etc.; we currently don't do anything with this information.

As a result of always using a TLS and a μTP connection in tandem, we had to define a simple protocol to match the two connection types: the first 48 bytes on a μTP connection sent from initiator to responder are the connection identifier which is simply the SSL session key of the corresponding TLS connection. This protocol is implemented in three functions:

- tor_tls_copy_master_key is used by both initiator and responder of a μTP connection to obtain the SSL session key of a TLS connection.

- channel_tls_send_utp_id is used by the initiator of a μTP connection to send the 48 bytes long connection identifier prior to any other data.

- tor_UTPOnReadProc, which was already mentioned above, is used by the responder of a μTP connection; it reads the first 48 bytes on a μTP connection, goes through the list of TLS connections, and finds the one that has the matching SSL session key.

The most important piece that is still missing is where Tor writes cells to μTP connections and where incoming cells received over μTP are handed over to Tor:

- channel_tls_write.*cell_method (three quite similar functions) write Tor cells to the outgoing buffer of a μTP connection and then call UTP_Write to tell libutp that there are bytes to be written; this is the second part of the additional level of indirection mentioned above: this call does not make libutp send out bytes immediately, but libutp will ask for the buffered bytes whenever it's ready to write them by calling tor_UTPOnWriteProc.

- run_connection_housekeeping sends PADDING cells over a μTP connection if one exists for a TLS connection.

- utp_process_cells_from_inbuf handles incoming cells that tor_UTPOnReadProc put into the incoming buffer of a μTP connection, but only when the TLS connection is done handshaking.

[4] Maybe we do the wrong thing here by calling UTP_Write if the connection becomes writable; maybe libutp can already handle that just fine?

- channel_tls_handle_state_change_on_orconn informs us that a TLS connection is done handshaking and calls utp_process_cells_from_inbuf to handle incoming cells that have arrived during the handshake process.

Last but not least we extended a maintenance function for µTP connections:

- run_scheduled_events calls libutp's UTP_CheckTimeouts once per second[5] which is vital for retransmitting UDP packets and generally keeping µTP connections alive.

3 Setup: Client and private bridge in public Tor network

The easiest way to test a new Tor datagram implementation is to use it only for the communication between two Tor nodes controlled by the tester. We came up with a setup consisting of a client and a private bridge. The client is configured to connect only via our private bridge. A private bridge is a bridge that doesn't publish its server descriptor to the bridge authority. The reason for using a private bridge in this setup is to avoid that other clients connect via it and mess with our experiment. This experimental setup is useful to test whether a new Tor datagram implementation works at all, and to do some basic debugging and tweaking.

Building our modified libutp and Tor sources is not as trivial as one would expect. That is why we decided to list detailed instructions here. The following instructions assume a freshly installed Ubuntu 13.10 as operating system. Instructions apply to both client and bridge:

```
sudo apt-get install build-essential git automake libssl-dev libevent-dev
mkdir src
cd src/
git clone https://github.com/kloesing/libutp
cd libutp/
git checkout -b utp origin/utp
make
cd ../
git clone https://git.torproject.org/tor.git
cd tor/
git remote add karsten https://git.torproject.org/karsten/tor.git
git fetch karsten
git checkout -b utp karsten/utp
```

The client's source can be compiled unchanged. But the *bridge's* source code must be changed to avoid connecting to any other relay via µTP. In order to do so, channel_tls_connect must be edited. Otherwise, the bridge will not be able to connect to the Tor network. The following code change is necessary:

```
diff --git a/src/or/channeltls.c b/src/or/channeltls.c
index 96055a3..ce68a9e 100644
--- a/src/or/channeltls.c
```

[5] Maybe this needs to happen much more often?

```
+++ b/src/or/channeltls.c
@@ -553,7 +553,7 @@ channel_tls_connect(const tor_addr_t *addr, uint16_t port,
   /* Create a uTP connection */
   tor_addr_to_sockaddr(addr, port, (struct sockaddr*)&sin, sizeof(sin));
   tor_addr_to_str(addr_str, addr, sizeof(addr_str), 0);
-  if (1) {
+  if (0) {
     log_info(LD_CHANNEL,
              "Trying uTP connection to %s", addr_str);
     tlschan->utp = UTP_Create(tor_UTPSendToProc, tlschan,
```

Now, both client and bridge source code can be compiled:

```
./autogen.sh
LDFLAGS="-L/home/ubuntu/src/libutp" CFLAGS="-I/home/ubuntu/src/libutp" \
    LIBS="-lutp -lrt" ./configure --disable-asciidoc --enable-gcc-warnings
make
cd ../
```

The relevant parts of the client's torrc file are:

```
SocksPort 9000
UseBridges 1
Bridge <ip-address-of-our-bridge>:9001
```

The relevant parts of the private bridge's torrc file are:

```
SocksPort 9000
ORPort 9001
BridgeRelay 1
PublishServerDescriptor 0
```

Figure 1 shows five runs of our client connecting via our private bridge over a µTP connection to the public Tor network. One of the bootstrap processes was quite fast, finishing in 30 seconds, but the others took over 100 seconds to complete. This plot is only supposed to show that the code is working and does not serve as performance comparison to current Tor.

While experimenting with this setup, we found and fixed a few issues in our implementation. We want to explain these issues briefly here to give some examples of the type of bugs that can be found in this setup:[6]

- Calling `UTP_CheckTimeouts` is more important for libutp than one would expect. We discovered a pattern where the sender sends just a single packet containing 1382 bytes to the receiver, the receiver processes the full Tor cells from it, and then they both sit there waiting until the next call to `UTP_CheckTimeouts`. Only then the receiver sends a 20 byte µTP control message to the sender which immediately sends another 1382 bytes before going silent again.

[6] Two more issues that may be worth mentioning: 2e0ac35: If a connection becomes writable, try to write (yet unclear if this actually fixed a bug); 43a013b: Process incoming cells on new connections immediately.

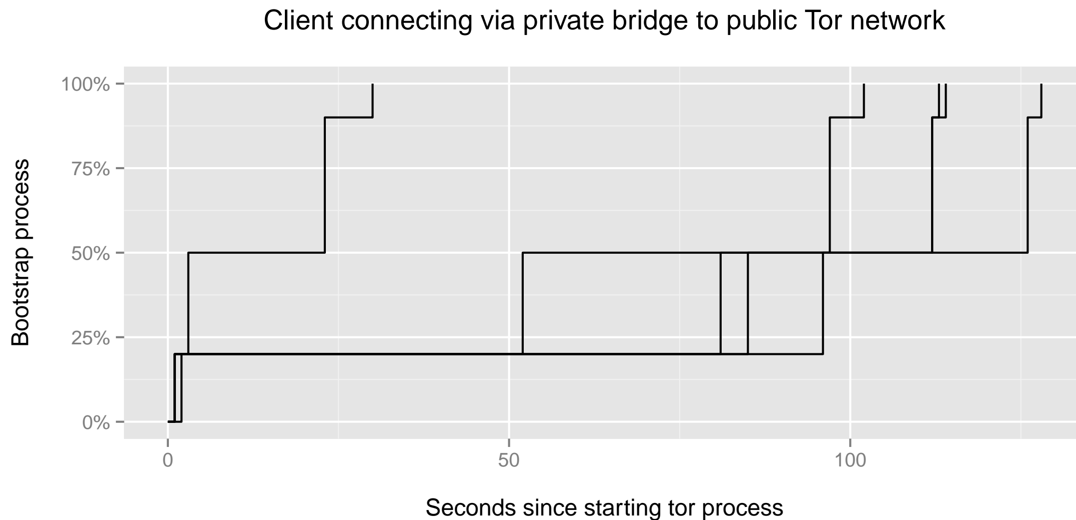

Figure 1: Bootstrap process of our client connecting via our private bridge over a μTP connection to the public Tor network.

- When libutp told us in `tor_UTPOnStateChangeProc` that a connection becomes writable, we didn't do anything with that information. Now we attempt to write in that case.[7]
- We previously missed a case when we received bytes on a new μTP connection that contained both a connection identifier and one or more full Tor cells. We should process the bytes exceeding the connection identifier immediately, because otherwise nothing might trigger processing them later.
- Unrelated to the datagram transport, we discovered that we must delete all `cached-*` files in the client's data directory *and* its `state` file before starting a new bootstrap process. Whenever we left the `state` file in place, the bootstrap process stalled for one minute which was unrelated to the datagram transport code.

4 Setup: Small private Tor network created by Chutney

A more advanced way to test a Tor datagram implementation is to set up a private Tor network and use the datagram transport for communication between most or even all nodes. Chutney makes it easy to generate torrc files for a small private Tor network that is supposed to run locally. This setup makes it somewhat harder to track down bugs, because all nodes are running the possibly broken Tor code, rather than just the initiator and responder of a single connection. The advantage of this setup is that it allows to test the new code on all different roles, including the role of a directory authority.

Running our implementation in Chutney is, fortunately, quite simple. We start by building libutp and Tor as explained in Section 3, but without applying the change for the private bridge. Then we clone Chutney and start a local Chutney network as follows:

[7] Actually, it's unclear if we actually *need* to do anything here, or if libutp can handle this just fine.

```
cd ~/src/
git clone https://git.torproject.org/chutney.git
cd chutney/
export PATH=~/src/tor/src/or:~/src/tor/src/tools:$PATH
```

Before running Chutney, we need to remove all torrc options which did not exist in Tor at the time of branching our code. The following patch should apply cleanly to commit 562b8f1:

```
diff --git a/torrc_templates/authority.tmpl b/torrc_templates/authority.tmpl
index 7bf99af..2fcb974 100644
--- a/torrc_templates/authority.tmpl
+++ b/torrc_templates/authority.tmpl
@@ -4,6 +4,5 @@ V3AuthoritativeDirectory 1
 ContactInfo auth${nodenum}@test.test
 ExitPolicy reject *:*
 TestingV3AuthInitialVotingInterval 300
-TestingV3AuthInitialVoteDelay 2
-TestingV3AuthInitialDistDelay 2
-TestingV3AuthVotingStartOffset 0
+TestingV3AuthInitialVoteDelay 20
+TestingV3AuthInitialDistDelay 20
diff --git a/torrc_templates/client.tmpl b/torrc_templates/client.tmpl
index 1eb1d99..be2c9dc 100644
--- a/torrc_templates/client.tmpl
+++ b/torrc_templates/client.tmpl
@@ -3,4 +3,3 @@ SocksPort $socksport
 #NOTE: Setting TestingClientConsensusDownloadSchedule doesn't
 #      help -- dl_stats.schedule is not DL_SCHED_CONSENSUS
 #      at boostrap time.
-TestingClientDownloadSchedule 10, 2, 2, 4, 4, 8, 13, 18, 25, 40, 60
diff --git a/torrc_templates/relay.tmpl b/torrc_templates/relay.tmpl
index 2f4b7f1..c2f8eb5 100644
--- a/torrc_templates/relay.tmpl
+++ b/torrc_templates/relay.tmpl
@@ -6,4 +6,3 @@ DirPort $dirport
 #NOTE: Setting TestingServerConsensusDownloadSchedule doesn't
 #      help -- dl_stats.schedule is not DL_SCHED_CONSENSUS
 #      at boostrap time.
-TestingServerDownloadSchedule 10, 2, 2, 4, 4, 8, 13, 18, 25, 40, 60
```

Once that is done, we can create and start a private Tor network:

```
./chutney configure networks/basic
./chutney start networks/basic
./chutney hup networks/basic
./chutney stop networks/basic
```

Figure 2 shows two clients connecting over µTP connections to a private Tor network created by Chutney. These bootstrap times are longer than in our previous setup, because

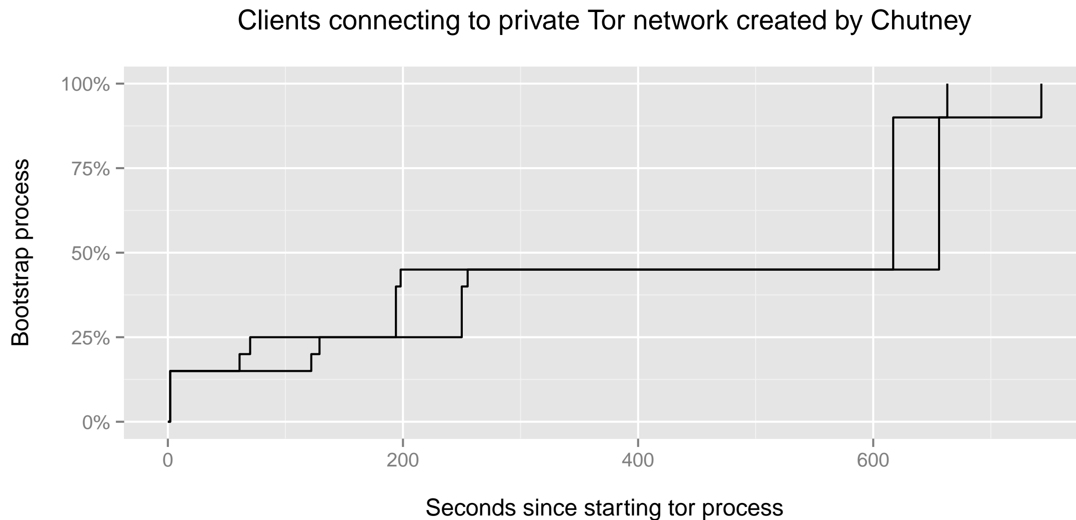

Figure 2: Bootstrap process of clients connecting over µTP connections to a private Tor network created by Chutney.

clients start at the same time as the directory authorities and relays, so there is no consensus yet and no way for clients to bootstrap immediately. But again, this plot is only supposed to show that the code is running, which is the case here.

It's worth mentioning that we found and fixed another bug using Chutney:

- Our earlier approach to create new connections to a remote relay was to first create the TLS connection and then the µTP connection. However, we only realized whether creating the TLS connection failed after creating the µTP connection. In that case, we'd free the TLS connection but not tell the µTP connection. This led to segmentation faults on directory authorities only which we would not have found in the earlier setup.

5 Setup: Large private Tor network using Shadow

The most sophisticated way to test new Tor datagram designs is to run them in large private Tor networks using the discrete-event simulator Shadow. These experiments not only reveal bugs, but can also provide performance results which can be compared to other datagram designs or to the original stream design. However, as indicated in the introduction, we do not focus on performance improvements here, because we found that our datagram implementation is not mature enough for such a comparison. A downside of using Shadow is that it adds even more complexity to the experiment and may introduce problems that are not present outside of the Shadow environment.

Building libutp, Tor, and Shadow is somewhat more complicated than setting up the earlier test environments. We again start by building libutp and Tor as explained in Section 3, but without applying the change for the private bridge. Next steps for building Shadow, including installing Clang/LLVM from source, are as follows:

```
sudo apt-get install cmake tidy libtidy-dev libglib2.0 libglib2.0-dev
wget http://llvm.org/releases/3.2/llvm-3.2.src.tar.gz
wget http://llvm.org/releases/3.2/clang-3.2.src.tar.gz
tar xaf llvm-3.2.src.tar.gz
tar xaf clang-3.2.src.tar.gz
cp -R clang-3.2.src llvm-3.2.src/tools/clang
cd llvm-3.2.src/
mkdir build
cd build/
cmake -DCMAKE_INSTALL_PREFIX=/home/ubuntu/.local ../.
make
make install
export PATH=~/.local/bin/:$PATH
cd ~/src/
git clone https://github.com/shadow/shadow.git
cd shadow
```

We need to make one change to Shadow to make it simulate our branch correctly: it needs to build the libutp source files itself, so that each virtual Tor instance has its own global libutp variables. The following patch should apply cleanly to commit 3b1a85e:

```
diff --git a/src/plugins/scallion/CMakeLists.txt b/src/plugins/scallion/CMakeLists.txt
index 8a6974f..2de3046 100644
--- a/src/plugins/scallion/CMakeLists.txt
+++ b/src/plugins/scallion/CMakeLists.txt
@@ -17,6 +17,8 @@ include_directories(${CURRENT_TOR_DIR}/src)
 include_directories(${CURRENT_TOR_DIR}/src/common)
 include_directories(${CURRENT_TOR_DIR}/src/or)
 include_directories(${CURRENT_TOR_DIR}/src/ext)
+include_directories(/home/ubuntu/src/libutp /home/ubuntu/src/libutp/utp_config_lib)

 include_directories(AFTER ${RT_INCLUDES} ${DL_INCLUDES} ${M_INCLUDES} ${GLIB_INCLUDES} ${EVENT2_IN

@@ -83,6 +85,7 @@ list(REMOVE_ITEM toror_sources
 #endforeach(headerpath)

 ## tor needs these defined
+add_definitions(-DPOSIX="1")
 add_definitions(-DLOCALSTATEDIR="/usr/local/var" -DSHARE_DATADIR="/usr/local/var" -DBINDIR="/usr/l
 remove_definitions(-DNDEBUG)
 ## disable warnings from tor code
@@ -90,6 +93,7 @@ add_cflags("-w")

 ## create and install a shared library that can plug into shadow
 add_bitcode(shadow-plugin-scallion-bitcode
+    /home/ubuntu/src/libutp/utp.cpp /home/ubuntu/src/libutp/utp_utils.cpp
     shd-scallion-plugin.c shd-scallion.c ${toror_sources} ${torcommon_sources} ${torextcurve_sourc
 add_plugin(shadow-plugin-scallion shadow-plugin-scallion-bitcode)
```

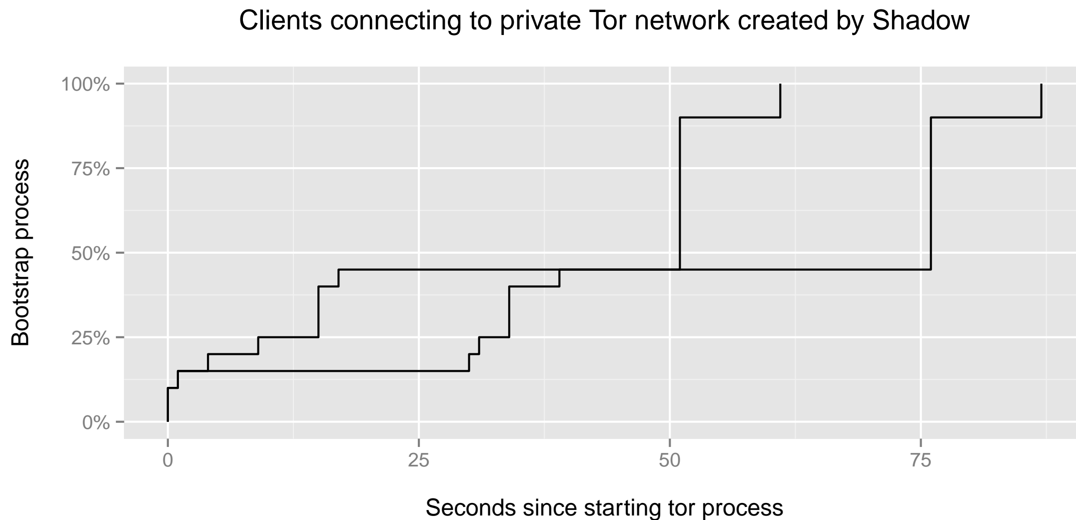

Figure 3: Bootstrap process of clients connecting over µTP connections to a private Tor network created by Shadow.

Then we run the remaining steps to build, install, and run Shadow:

```
./setup dependencies
./setup build -g -i /home/ubuntu/src/libutp \
    -i /usr/include/x86_64-linux-gnu/c++/4.8 -l /home/ubuntu/src/libutp \
    --tor-prefix /home/ubuntu/src/tor --tor-lib utp
./setup install
export PATH=~/.shadow/bin/:$PATH
cd resource/examples/scallion/
tar xf minimal.tar.xz
cd minimal/
scallion --log-level=INFO
```

Figure 3 shows two clients connecting over µTP connections to a private Tor network created by Shadow. Once again, this plot is only supposed to show that the code is running.

Here are a few pitfalls from testing our datagram implementation using Shadow, again meant as examples for future tests:

- We should register an `EV_WRITE` event with libevent whenever we want to write to a UDP socket, rather than writing to the socket directly. While this doesn't matter as much when running our implementation outside of Shadow, Shadow cares about this detail. It's also the better design.[8]
- Apparently, the `do_main_loop` function is never called in Shadow, so our callback that we added to check µTP timeouts more often than once per second was never registered with libevent. In general, it's a good idea to add log statements to all changed code, and make sure it gets executed as expected.

[8]However, it's still unclear whether this actually solved a bug.

- We found that Shadow had a bug[9] where UDP sockets that bind to `INADDR_ANY:port` later got their port changed. They still received packets from the initial bound port, but packets they sent appeared to come from the changed port.

- libutp uses a few global variables, including a list of open μTP connections. In the case of a single Tor process using libutp as a library, this works just fine. But in case of Shadow, the global variables are shared among all Tor instances, which leads to chaos very quickly. The fix is to build libutp as part of the scallion plug-in to ensure that Shadow will hoist the libutp variables and make a copy of them for every virtual Tor node.

6 Conclusion

One conclusion from experimenting with our libutp-based datagram transport implementation is that it's far from trivial to replace the transport in Tor. And we only looked at the allegedly simple case of using the datagram transport in a separate (part of a) Tor network.

The next step will be to debug our implementation enough to compare its performance to an unchanged Tor version. It may be that there are still bugs in our code, or it may be that we need to tweak libutp's parameters to make it more suitable for Tor. We may also want to rethink our design decision to always open both a TCP connection and a μTP connection in parallel, though this should not affect performance results much. We might also want to switch from a single μTP connection between client and relay or between two relays to one μTP connection per circuit. Performance evaluations should also include setups with additional TCP load in the network to see how much μTP yields to TCP and how that degrades Tor performance.

More steps further down the road are to try out other datagram transports than libutp and to think about deploying a new transport in the existing Tor network. We have still got a long way to go.

Acknowledgments

Nick Mathewson, Sebastian Hahn, Matthew Finkel, and Yawning Angel provided valuable feedback and helped out with code fixes while writing this technical report.

References

[1] Steven J. Murdoch. Comparison of Tor datagram designs. Technical Report 2011-11-001, The Tor Project, November 2011.

[2] Steven J. Murdoch. Datagram testing plan. Technical Report 2012-03-002, The Tor Project, March 2012.

[9] https://github.com/shadow/shadow/issues/180