mPerf

*: Current version(1.2.0130 pre-Release) is a **pre-Release** version, which means:

- The programs are in **Release** state and marked *Release*(use -v to view).
- We are still working on other things like program manual, installation instructions, etc. But you can start using the programs now if you know how to make them work.
- **You may not distribute this packet since copyright information can be incomplete!**

**: We are building version 1.3 now. We plan to use UDP for configuring since TCP handshaking is very expensive in high-RTT circumstances.

<!---
is a **Beta** version, which means:

- This is a **ready-to-use** version!
- We have fixed all fatal and severe bugs we have known, but we are still doing tests and further optimizations.
- The calling interface will NOT be modified until next **Release** version.
- **You may not distribute this packet since copyright information can be incomplete!**
  -->

<!---
is an **Alpha** version, which means:

- This program is not stable and may do harm to your computer!
- Some severe or fatal bugs in the program are not fixed yet.
- We may modify the calling interface until next **Beta** version.
- **You may not distribute this packet since copyright information can be incomplete!**
  -->

`mperf` is a `iperf`-like tool that enables you to use TCP or UDP protocol to send or receive some data over network. Unlike `iperf`, `mperf` is specially designed for *tough* networks with high RTT, high loss rate, high out-of-order delay, etc, it works robustly and never gives unexpected outputs under those circumstances. We designed this tool in order to provide a way to measure the performance of those networks in an all-non-interactive and robust way.

References

1. iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool
   We use same sending block size for TCP tests as `iperf`, and we also use `netdial` function from `iperf` to establish connections(it originally comes from Libtask).
2. Computer Systems: A Programmer's Perspective, 2/E (CS:APP2e) (example code)
   We use `open_listenfd` wrapper function from `csapp.c` to listen on a specified port, also our data sending&receving functions and sigaction() wrapper function are inherited from `rio_writen`, `rio_readn`, `Signal` in `csapp.c`.

Install & Uninstall

`mperf` works on Linux only.

- Linux:
  install: `make && make install`
  uninstall: `make uninstall`

Using the programs & scripts

- `mperf-client`
  The client program, used for receving data.
  Run `mperf-client -h` for detailed information.
- `mperf-server`
  The server program, used for sending data.
  Run `mperf-server -h` for detailed information.
- `mperf-udpsender`
  The program used for testing the network condition with UDP, it sends small UDP packets periodly.
  Run `mperf-udpsender -h` for detailed information.
- `mperf-udpreceiver`
  The program used for testing the network condition with UDP, it receives packets from `mperf-udpsender` and then sends them back to it.
  Run `mperf-udpreceiver -h` for detailed information.
- `mperf-kill`
  List and kill all running `mperf` programs.

Examples

Let's start from launching a server program with default configuration.

`mperf-server -p 20001 -l mperf-controller.log -L mperf-server.log &`

This will start the controller program, `mperf-server` will listen on port 20001(20000 + 1).

Then try to connect to the server!

`mperf-client -c 127.0.0.1 -B 127.0.0.1 -p 20000 -t 1`

This will start a client program trying to connect to the receiver launched by `mperf-runserver` on `127.0.0.1:20000`. It will use the local network interface(`127.0.0.1`) to receive data, and keep receiving for 1 second.

The output can be something like this:

```
[ Message ](      0.000206): mperf log "1.1.0098-Release"
[ Message ](      0.000247): Timestamp initialized at +0800 2017-08-16
19:55:44(1502884544.876773)
[ Message ](      0.001445): Connection established, mss is 21845.
[ Message ](      1.001575): EOF reached.
[ Message ](      1.001602): Test summary:
[ Message ](      1.001609): ->Total time: 1.000128s
[ Message ](      1.001616): ->Bytes received: 8606318592
[ Message ](      1.001623): ->Bandwidth: 8605217124.208101Bytes/sec

[ Message ](      1.001631): Transfer complete.
```

The summary part shows the amount of data you have received, the total time elapsed and the average bandwidth. The log also shows the MSS of this connection and the value of startup timestamp(see Logging(not available now!) for details).

Also, we can also choose to transfer a small amount of data, like 1K:

`mperf-client -c 127.0.0.1 -B 127.0.0.1 -p 20000 -n 1024`

You can get those outputs:

```
[ Message ](      0.000189): mperf log "1.1.0098-Release"
[ Message ](      0.000228): Timestamp initialized at +0800 2017-08-16
19:57:34(1502884654.597902)
[ Message ](      0.001227): Connection established, mss is 21845.
[ Message ](      0.001374): EOF reached.
[ Message ](      0.001396): Test summary:
[ Message ](      0.001406): ->Total time: 0.000130s
[ Message ](      0.001418): ->Bytes received: 1024
[ Message ](      0.001428): ->Bandwidth: 7876923.076923Bytes/sec
[ Message ](      0.001455): Transfer complete.
```

After the tests, let's have a look at `mperf-controller.log` and `mperf-server.log`. These files contains information of formal tests, all in the server's side of view:

```
# mperf-controller.log
[ Message ](      0.000247): mperf log "1.1.0097-Release"
[ Message ](      0.000294): Timestamp initialized at +0800 2017-08-16
19:55:21(1502884521.500100)
[ Message ](      0.000354): Listening on port 20001.
[ Message ](     23.377200): Listening on port 20001.
[ Message ](     23.377217): Connected with 127.0.0.1:53091
[ Message ](     23.377250): Trying to reconfigure the server...
[ Message ](     23.377295): Reconfiguring, control message is "0 1 -1".
[ Message ](     23.377665): Server(11166) starting...
[ Message ](     23.377973): Reconfigure completed.
[ Message ](     23.378034): Connection with 127.0.0.1 closed.

[ Message ](     23.378049): Listening on port 20001.
[ Message ](     24.385141): --Child process(11166) terminated
[ Message ](    133.098199): Listening on port 20001.
[ Message ](    133.098249): Connected with 127.0.0.1:52661
[ Message ](    133.098270): Trying to reconfigure the server...
[ Message ](    133.098291): Reconfiguring, control message is "1 210 1024".
[ Message ](    133.098613): Server(11181) starting...
[ Message ](    133.098874): Reconfigure completed.
[ Message ](    133.098923): Connection with 127.0.0.1 closed.

[ Message ](    133.098932): Listening on port 20001.
[ Message ](    133.099432): --Child process(11181) terminated

# mperf-server.log
[ Message ](      0.000094): mperf log "1.1.0097-Release"
[ Message ](      0.000130): Timestamp initialized at +0800 2017-08-16
19:55:44(1502884544.877844)
[ Message ](      0.000148): Trying to reconfigure server(11166).
[ Message ](      0.000168): Reconfigured with type = long, len = 1
[ Message ](      0.000198): Listening on port 20000.
[ Message ](      0.000420): Connected with 127.0.0.1:57580
[ Message ](      1.000469): Long test summary:
[ Message ](      1.000484): ->Bytes transferred: 8606318592
[ Message ](      1.000487): ->Time elapsed : 1.000019s
[ Message ](      1.000490): ->Bandwidth: 8606155075.053574Bytes/sec
[ Message ](      1.000506): Connection with 127.0.0.1 closed.

[ Message ](      0.000059): mperf log "1.1.0097-Release"
[ Message ](      0.000078): Timestamp initialized at +0800 2017-08-16
19:57:34(1502884654.598794)
[ Message ](      0.000094): Trying to reconfigure server(11181).
[ Message ](      0.000114): Reconfigured with type = fix, timeout = 210, size =
1024
[ Message ](      0.000152): Listening on port 20000.
[ Message ](      0.000346): Connected with 127.0.0.1:58530
[ Message ](      0.000405): Fix test summary:
[ Message ](      0.000418): ->Bytes to transfer: 1024
[ Message ](      0.000428): ->Bytes transferred: 1024
[ Message ](      0.000438): ->Bandwidth: 31030303.030303Bytes/sec
[ Message ](      0.000451): ->Time elapsed : 0.000033s
[ Message ](      0.000482): Connection with 127.0.0.1 closed.
```

This is the basic usage of `mperf` TCP tester programs. You can run the programs with `-h` option to get further information.

Have fun!

Philosophy & principles

This part is for ones who want to get an insight of our programs quickly.

The program follows the principles described below:

- Robustness comes first.
  We ensure that the programs give expected outputs and server program always in a valid state(ready to do new tests).
  - All-in-log.
    We handle exceptions properly to give expected outputs. The log thus contains all information about the events happened when the programs are running. Whenever a system call returns with an error, or some signal(like SIGALRM) was received, we will write logs.
  - Preemptive testing.
    We don't allow concurrent tests. So when a new test request comes, we must choose to decline or to end the previous test. We choose to ensure that we can always start new tests. Consider that we passed wrong arguments in a previous test, like -t 1000000, if we choose to end the previous test, we won't have to terminate the server to start a new test. In this way we will be able to fully control the test flow from client side.

- Synchronize *at design time*
  In order to improve robustness, we reduce runtime information shared by client and server as much as possible. We notice that to handle a test properly, the client only need to know if it can connect to the server and if the server can do the test specified by the user, and the server only need to know the test arguments. After that, the client can simply connect to the server, receive data until the timeout threshold is reached, or it received enough data, then simply abandon the connection and terminate. As for the server, simply send data as

requested, then abondon the connection can be just fine. This reduces synchronization cost and in this way, after communicating the arguments, the client and server can work without knowing each other's working progress.

  - Avoid errors to survive.
    We have a controller process on the server side. It's an iterative server who accepts test arguments continuously. Everytime a pack of arguments is received, it use `fork()` to start a new server process to do the test, then return to listening state. The server process then send data to the client. The key idea is, let the controller process use network as less as possible, thus it doesn't need to deal with any potential errors that can occur during the real test. Whatever happens, the controller will be in a valid state when next test request arrives, we can then simply kill the previous server process and start a new one. This ensures that the server program always in a valid state.

- As fast as possible.
  As a measuring tool, we want to fully utilize the network bandwidth as a user-space program. The ideal situation is, we do nothing other than calling `write()` or `read()` in the programs, but sadly that's impossible. :(But anyway, we are working to achieve that goal approximately.
  - System calls as last solution.
    We are using system calls as less as possible to improve the performance of the programs in order to utilize the network bandwidth better(We are also using a sending loop as simple as possible to send more data).
    For example, we always use atomic operations instead of mutexes.
  - Application-layer data only.
    As for the output, we only show the application-layer data.
    We notice that when measuring the performance of networks, we need more than application-layer data to know about metrics like loss rate, time series of throughput, etc. Some of them can be acquired by using system calls like `getsockopt()`, but we choose not to use them here according to "System calls as last solution" rule. You can use packet capture tools like wireshark to know about what happened on layer 1~4.
  - One process per task.
    We prefer to use more processes to deal with more than one tasks(instead of using threads) to avoid any synchronizing overhead. So we are using fork instead of thread pool to implement the server.