

## **xxxTap – Middleman for Inter-Process Communication data**

**tcpTap.c** - allows the user to interactively intercept, display and/or alter client-server TCP stream socket data.

**udpTap.c** allows the user to interactively intercept, display and/or alter client-server UDP datagram socket data.

**webTap.c** allows the user to interactively intercept, display and/or alter client-server data between a web browser and a web server.

What follows describes tcpTap, but udpTap and webTap are designed to the same exact interactive user specifications.

**Usage:** tcpTap port\_number host\_name [log\_file]

port\_number: the server's port number

host\_name: the server's host name

log\_file: an optional data log

This program acts as a middleman, running on one of three network nodes. A client runs on a second node, and the server runs on a third. (Note that tcpTap connects to the server, and the client must connect to tcpTap on the same port. That means tcpTap must run on a node that is different than that of the server. The client, however, may run on the same node as either tcpTap or the server, but the operation is easier to visualize on three separate network nodes.) The client connects to tcpTap, which connects to the server. tcpTap allows interactive inspection and manipulation of tcp stream data sent from the client to the server and back, over the connection port number and host name specified on the command line. In most cases you should be able to debug a client-server problem without a hard copy log, but tcpTap accepts an optional third command line operand that becomes a time-and-date-stamped data event log, for further, more detailed examination and analysis when necessary.

### **Installation:**

Create a subdirectory containing midMan.tar and issue

```
> tar xvf midMan.tar
```

```
> make [all]
```

Three new binary executables named tcpTap, udpTap and webTap will be created in the same subdirectory.

### **Execution:**

This example assumes you have a client, named myclient, and a server, named myserver, and that they communicate over a tcp stream socket on port number 5678. It also assumes that you have three nodes on your network, named alpha, beta, and gamma.

1. Start myserver on gamma
2. Start tcpTap on beta, as follows
  - > tcpTap 5678 gamma
3. Finally, start the client on alpha
  - > myclient beta

Note that you may have many command line operands. It is important that you substitute the hostname "beta" for the operand that tells myclient where the server, myserver, is running.

At this point, myserver is running on node gamma, and tcpTap is connected to it, as if it was myclient. The server myclient is running on node alpha, and it is connected to tcpTap, as if it was myserver.

Each time myclient sends a request to myserver, tcpTap intercepts it and forwards it to where myserver is actually running. When myserver sends a response back to myclient, tcpTap again intercepts it, and forwards it back to where myclient is actually running.

The tcpTap process acts as a "middleman" between your client and server, and neither is the wiser. You may now ask tcpTap (through its commands) to "watch" or "edit" the tcp data stream in either or both directions.

You may also "inject" data destined for either the client or the server process. Type "h" at the keyboard on beta and tcpTap will display a list of its commands along with a brief description of each.

Messages may optionally be logged to a data file whose name is specified on the command line. Each message is time and date stamped and displayed in hex along with their corresponding ASCII characters.

**Examples:** Here is the startup for all examples – time flows from top to bottom:

```
alpha          beta          gamma
                > tcpSer
                > tcpTap 6556 gamma
> tcpCli beta
                Listening for client on port 6556...connected
                Connecting to gamma server on port 6556...connected
                tcpTap: Command?
```

From here on, all tcpTap user commands (in bold) and their responses will be on **beta**.

**Example #1:** Watching data from the client to the server:

```
tcpTap: Command? w
tcpTap: From [client|server|both] (c|s|b): c
tcpTap: Command?
```

[Here the client sends "3 + 8" to the server.]

```
Client message buffer:
0000: 33202b20 380a00      [3 + 8.....]
```

[Each message received from the client will interrupt any interactions. Note that if you enter **s** instead of **c** the *response* data is dumped out. Finally, a **b** dumps messages from both directions, as you might expect.]

**Example #2:** Clear all watches and edit requests:

```
tcpTap: Command? c
All watch and edit requests cleared.
tcpTap: Command?
```

**Example #3:** Modifying data from the client, intended for the server:

```
tcpTap: Command? e
tcpTap: From (client|server|both) (c|s|b): c
tcpTap: Command?
```

[At this point, "2 + 4" is typed at the client]

```
Client message buffer:
0000: 32202b20 340a00      [2 + 4.....]

edit: Command? c
edit: Enter hex buffer offset: 0
edit: Enter hex buffer data for 0x00: 34
0000: 32202b20 340a00      [4 + 4.....]
edit: Command? q
edit: Use buffer contents? y
tcpTap: Command?
```

[The server receives "4 + 4" and responds "4 + 4 = 8" to the client.]

**Example #4:** Injecting data (from the client) to the server:

```
tcpTap: Command? i
tcpTap: Enter maximum buffer size: 7
tcpTap: To (client|server) (c|s): s
tcpTap: Command?

Server message buffer:
0000:  20202020 202020          [          .....]
edit: Command? c
edit: Enter hex buffer offset: 0
edit: Enter hex buffer data for 0x00: 37202d20320a00
0000:  32202b20 340a00          [7 - 2.....]
edit: Command? q
edit: Use buffer contents? y
tcpTap: Command?
```

[The server receives "7 - 2" and responds "4 + 4 = 8" to the client.

You will have to hit <enter> at the client to see this response, as the client is still in *input* mode. (Hitting enter may not work for UDP data because the incoming datagram packet is discarded at the client.)

The **e** (for **edit**) command is similar to the inject command. If you request an edit of client data, for example, and the incoming contains "3 \* 5" you may change the 0x2a (the asterisk at buffer offset 0x2) to a 0x2b (a plus sign). The server will receive "3 + 5" and respond back with a "3 + 5 = 8" string.

If a bug causes your client to build a bad date field, for example, it's faster to catch and fix the data on the fly, fixing the client source code later after you've determined that the bad date was the only cause of the abnormal behavior under test.

Remember that you may type **h** (or **help**) at any command prompt to get a list of supported commands and a short description of what they do. Enter **man ascii** at any Linux command prompt to get a listing of the hex values for all of the characters.

Note that **udpTap** and **webTap** support the same command set for UDP and HTTP inter-process communications, respectively. Recall that UDP packets may be discarded on error. And HTTP testing requires an active web server. We run Apache on Linux and put a simple web site, including an **index.html** file, under /var/www/html/.

Above all, we here at DSRLab hope you will gain some benefit from these products, learn a few things, and have some fun along the way!