The Herculean Effort

Fake it till you make it!

I am composing this guide as a collection of my progress on optimizing the Hercules mainframe emulator, and getting all of the networking facilities to work. Starting in 2020, I undertook serious efforts to profile and fix performance issues in Hercules, and I constructed what I believe is a machine that legitimately may be a serious competitor to mainframes for hobbyists. Though I am not a mainframer by trade (neither is the person that helped me write this), I have recorded progress and composed this document.

List of Topics

1. Selecting the right CPU and RAM
2. Optimizing storage arrays
3. A SAN?
4. Hercules networking
5. Making guests run
6. Hercules disks
7. ICC emulation
8. SNA

I would also like to recommend to any and all readers who may find this that you should not rely upon our findings to produce serious production-grade systems. We are hobbyists, we are building a hobbyist system. Both of us have limited server storage space (though my counterpart appears to have about 3 server racks…) so compactness and performance density is a must. Under no circumstances should you violate real license agreements in the "real world" by emulating real licensed mainframe workloads on a system they are not licensed to run on. With that being said, this particular system is of terrific interest for the sole purpose of modernization. Of important interest is that other systems that are considered legacy systems have moved over to emulations. The Unisys mainframes running both OS 2200 (Unisys proper) and MCP (Burroughs acquisition) are both emulator-boxes. The Bull Sequana machines that run GCOS7 and 8 (as well as Multics) are emulators. Put simply, emulation seems to be the future and we wish to secretly introduce it. As my assistant in this project famously said, "a suitably-accurate emulation should be indistinguishable from the original."

Selecting the right CPU and RAM

Before we travel down into the depths of deep system optimization, one must consider what an ideal machine would look like. During the original development sprint, two machines were designed:

1. Intel Core i9-10900K, DDR4-2933 64GB RAM, Intel Z490 chipset, 4 Samsung 970EVO NVMe SSDs
2. AMD Ryzen 9 3950X, DDR4-2933 64GB RAM, AMD X570 chipset, 4 HP 4XE500 NVMe SSDs

I elected to build both machines – both have been built at the time of writing this and yield similar performance. I mounted my machines in Rosewill 19-inch rackmount server cases, and they can be observed in a server rack under a 40 gigabit Ethernet switch (we will be discussing SAN technologies later). Cooling on these two machines was achieved in the same way: a Corsair H115i AIO closed-loop watercooling system was easily installed within the systems and yielded very good performance. To facilitate the mount within the 19-inch cases, I

attached the radiator near the top of the rear of the case (which featured a large vent) by using two 140mm watercooling radiator mounts I brazenly sawed out of a desktop PC case; it was then "hung" from the top lid.

Thermal output tends to be linear with system load. Under no load (which is not a workload seen in this project), the CPU would idle at about 43 degrees (note for future reference, all thermal measurements will be in C, as this is the standard for measuring silicon die temperatures – 100 degrees C is the maximum any CPU of the modern design parameter will function at); under idling load (about 25% total system usage sustaining many emulated guests), the CPU measured 58 degrees; the CPU peaked to 74 degrees under 100% usage. RAM temperature was not a concern. In the Intel system, it did not have any storage HBA controller cards, but I had both a high-end SAS RAID controller card and a 40 gigabit Ethernet card (these outputted so much heat that they required a fan to be mounted above them) in the AMD system. The NVMe SSDs were mounted directly onto the motherboard, and the heat gets piped to a heat exchanger at the bottom right of the motherboard (on both the MSI AMD and Gigabyte Intel motherboards).

Initial benchmarks yield similar results, but the C compiler options used greatly resulted in performance shifts. Other languages were not benchmarked as they are not of interest (because Hercules, QEMU, and simh are not written in anything but C). The primary bottleneck (term for a limiting factor) is the CPU clock speed – not the amount of CPU cores as mainframes are mostly a single-threaded application until you are running a mainframe application "en masse" (this was the point where the AMD machine won out over the Intel machine, as it had more cores).

Optimizing storage arrays

Before we discuss how one would design an ideal storage subsystem for our virtframe, we must consider what we need: RAID. There are two ways to achieve this: hard RAID and soft RAID. The former is often seen accompanying SAS or FC (fibrechannel) disk arrays, and the latter is often seen with NVMe arrays. We must also discuss the differences between NVMe and a traditional disk – a traditional disk (or an SSD with a SATA or SAS interface) presents an emulated hard disk to a disk controller that thinks it's talking to a hard disk controller. You "seek" the drive's emulated disk head, and you read and write data. NVMe, on the other hand, is accessed like a flash storage memory array (and is therefore somewhat like RAM) – you specify an address, it appears within a memory "window" (to avoid having to burn a lot of RAM addresses for possibly petabytes of flash memory), and you can transfer data to and from the SSD by reading and writing this DMA window. No SCSI or ATA commands are present with NVMe, unlike regular SSDs. Also, the NVMe interface runs at native PCIe bus speed (since NVMe is an extended PCIe bus protocol); SATA SSDs are generally limited to 6gbps and SAS SSDs are generally limited to 10gbps. I only used NVMe SSDs mounted in M.2 slots on the motherboard (found between the PCIe slots), but my counterpart opted to use a mixture of three storage technologies (10000RPM WD Black hard disks, 500GB old SLC flash SATA SSDs, and 500GB NVMe SSDs).

RAID, as mentioned, is nearly totally paramount for increasing disk performance. RAID 0 is very fast, but a single disk failure will result in a total loss of data. RAID 5 is decently fast, but CPU time must be burnt up computing parities – the result of this is that a single drive can fail and all of the data can be recovered. Hard RAID generally refers to RAID provided by some kind of real disk controller card, and soft RAID generally refers to a RAID-style array simulated by an operating system. It should be noted that there is no standard for hard RAID arrays – that is, a RAID array made on an LSI controller will not work on a BusLogic or IBM controller. Soft RAID, on the other hand, is portable – a real RAID controller card can be set to single-drive mode (in fact, this is

the industry standard) then a RAID array is assembled on top of that by using an OS's soft RAID facility; Essentially every OS that exists today provides some form of a soft RAID facility, you are on your own to assemble an array, though:

Linux: MD and LVM
Windows: Striped volumes
FreeBSD: GEOM RAID mode
NetWare: striping

When using NVMe SSDs, hard RAID may appear to be possible, but it is actually merely an illusion! You may notice options for NVMe RAID in your BIOS settings (Intel Rapid Storage Technology or AMD RAIDXpert), but this is fake RAID. There is no hardware RAID acceleration like you would find on a conventional RAID HBA (host bus adapter) card: instead, the presence of some SATA controller is presented as a fake device by a program that runs on the CPU in system management mode. Though the OS "sees" a SATA controller synthesized from an emulated NVMe array (how strange is that?), there is no silicon converting a multi-pathed NVMe interface to SATA. As such, using soft RAID *in the operating system* is ideal as context switches do not have to occur to drive the emulated SATA controller.

A SAN?

A Storage Area Network may be advisable for some users that have exhausted their PCIe slots in their virtframes (you should, no doubt, be using NVMe SSDs). You can erect an external box that may not have much CPU power, but features a terrific amount of PCIe bus lanes. Upon doing this, you can insert a fibre channel compatible card in both the server and its associated client. You can use proper FC cards, but these are more expensive than FCoE (fibre channel over ethernet) cards as they require a special FC switch (which inevitably has a bunch of licensing headaches that you can't crack your way out of). FCoE with a Mellanox card (such as a dual 40 gigabit ethernet ConnectX-3) is much cheaper. When the card operates in FCoE mode, it presents an emulated FC controller to the host that physically speaks Ethernet signals.

SANs served from FreeBSD are very performant, as ctld provides all of the required transports. Linux has LIO (LinuxIO) for the client work – researching this on your own will be most advisable if you wish to use a SAN. If you do not wish to use a disk-type SAN, you can do what Evie often does and have a file server stuffed with NVMe sticks (so-called because they are the size of sticks of gum) that is then in-turn accessed from a network of remote servers using NFS or SMB. I went with a proper disk SAN solution where LUNs are exported and accessed on a remote system, but a "file server" solution may be more flexible (and, most importantly, more than one host can access a single storage extent at the same time because you are sharing files, not entire disks).

Hercules networking

Networking with Hercules is rather interesting. Before we discuss this, we must discuss networking on mainframes in general. In the early days, the only form of networking that you had was either a 2703 serial port controller (later emulated by the 3705 with ease), or a 3088 CTCA (Channel to Channel Adapter). The 2703/3705 provided serial port links (possibly over modem links) that could run RJE and NJE (NJE being an actual useful network protocol). The 3088 coupled two mainframes together and allowed them to communicate by writing to or

reading from a device channel. At first, a 3088 was seen as a more performant alternative to a 2703 – the 2703 ran in the bauds range (at max with a 3705, 115200 baud), whereas the 3088 ran at a full block multiplexer channel speed 4.5 mega*bytes* a second). We can emulate "pure" 3088 CTCAs with Hercules, and we can establish multiprotocol links over these (VM/CMS's PVM, JES2/JES3/RSCS/POWER NJE, TCP/IP links *(see below)*, VTAM CTC and MPC links, etc).

Eventually, Ethernet and Token Ring were invented. In response, a variety of LAN interfaces were constructed. At first, a channel-attached IBM PS/2 desktop called a 3172 was wired up to the mainframe – this was called a "LAN Channel Station." There was a similar device where IP packet processing was done on the PS/2 or RS/6000 (POWER UNIX server) called Combined Link Access to Workstation (CLAW), and both the IBM 9370 and original ES/9000 line featured an integrated communications controller that emulated a 3172. However, the 3172 was fundamentally an 3088 CTCA with the remote side being a PC. Nonetheless, by the time these 3172s had ran their course, they had developed support for a variety of protocols:

- TCP/IP – this required two channels, as it ran in full duplex
- SNA – this attached to VTAM, but ran in half duplex (but was not slow)
- OSI – since the 3172 sent raw Ethernet datagrams over the CTCA, both TCP/IP and OSI packets could be transceived on an Ethernet network (token ring could not run OSI)

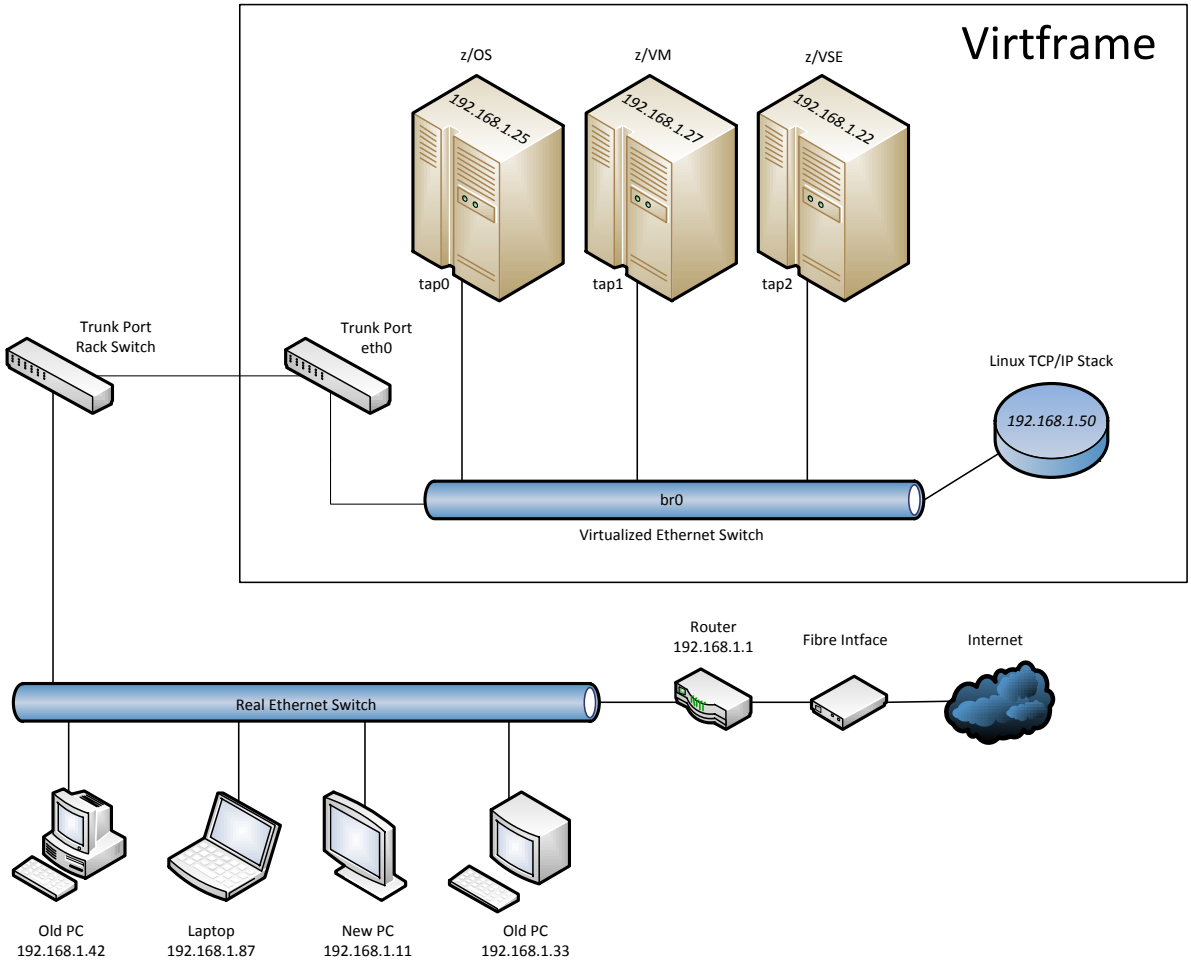
The LCS (3172) was eventually replaced with a device called an Open Systems Adapter (though it originated as an integrated communications controller on an ES/9000). The OSA emulates an LCS, but newer ones provide a different protocol. The OSA-1 on the ES/9000 was just a network card, but it would eventually (by the time the OSA-Express2 came out) be a core part of the mainframe. The OSA-2 provided an Ethernet, Token Ring, or FDDI interface controller (well, more accurately, several), but was eventually replaced by an OSA-Express – this was, eventually, replaced with an OSA-Express2. The OSAX2 supports the following operating modes:

- OSE – LCS emulation mode (SNA or "layer 2" aka TCP/IP and OSI)
- OSC – the ICC (Integrated Console Controller), a local 3270 controller that provides a TN3270 interface
- OSD – QDIO mode, new on the OSA-Express, layer-2 or layer-3
- OSN – an OSA under the control of an emulated NCP (Communication Controller for Linux)

One might ask what QDIO is! Well, when the LCS started to appear slow in the age of gigabit networking (because it was limited by interrupt generation and the performance of a free CCW channel), someone at IBM decided to design a memory-mapped-I/O network card that worked somewhat like the high-performance network cards in PC servers. These new OSD-mode OSA-Express cards were radically different (but retained compatibility) than the OSA-2. QDIO mode can run in two different modes: layer-2 mode and layer-3 mode. Layer-2 mode is, if you are versed in networking, Ethernet. Layer-3 is the protocol that rides on top of Ethernet, which, in this case, refers to IP. An OSD card running in layer-2 mode will allow Ethernet switch bridging (useful in z/VM with VSWITCHes), but a card running in layer-3 mode can only speak IPv4 and IPv6 (but 14 bytes are saved with every packet, as an Ethernet header need not be transmitted from the CPU to the OSAX). Hercules supports all of these, but, in the sense of sanity, we will be using layer-2 devices as much as possible.

Remember, LCS 3172s aka OSEs are layer-2 devices (except in SNA mode, which we will cover later).

Since we are running Linux on a Hercules host (FreeBSD is difficult for other reasons we will not discuss), we will need to erect some form of network. Study the following diagram:



While this may appear to be somewhat complex at first, let's break this down. For one, the netmask of this network is 255.255.255.0, which means all the hosts are on the same logical LAN. An Ethernet datagram can pass from the z/VSE system straight to that new PC, and its contents will never be altered by a router (that is, a router will alter MAC addresses as it passes from network to network). If you are unfamiliar with IP subnetting, see the back of this document (I'm not going to give a full tutorial on subnetting, just provide a pretty picture that gets you the gist of it). So, what we have here are three virtual mainframes attached to a virtual Ethernet switch within the server, with Linux's TCP/IP stack attached to the virtual switch. The switch has, in this picture, five ports:

- z/OS – tap0
- z/VM – tap1
- z/VSE – tap2
- Trunk port – eth0
- Linux IP stack – IP address assigned to the br0 interface

Linux will access the network itself by binding sockets to the IP address assigned to the br0 interface – this is basically a virtual "port" on the virtual switch. The trunk port will allow Ethernet datagrams of *any* protocol (IPv4, IPv6, IPX, DECnet, SNA, NetBEUI, OSI, Apollo, XNS, AX.25, VINES, AppleTalk, CDP, spanning-tree, etc)

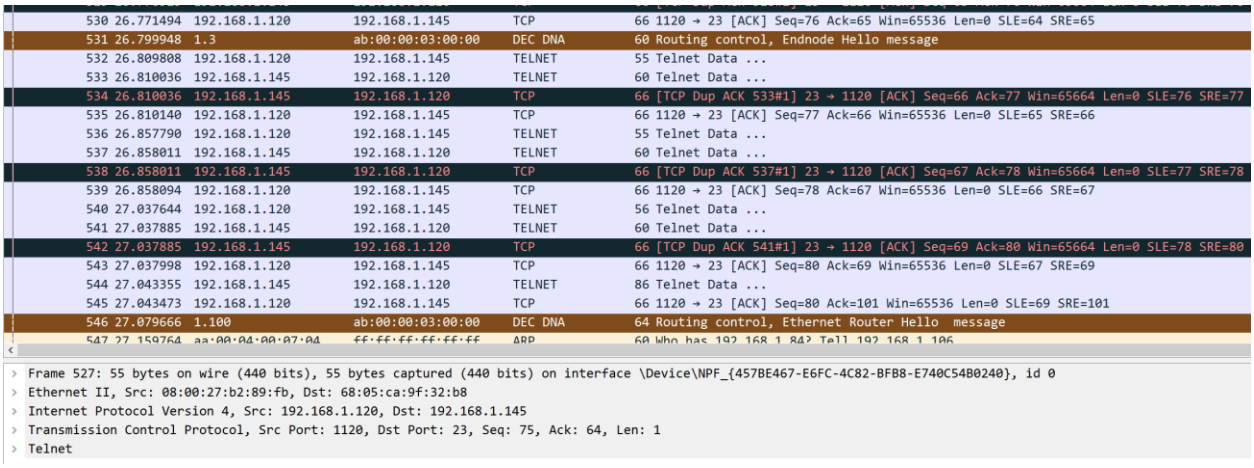
to pass in and out of the server. If a DECnet router announcement packet emits from the router in the picture, all the mainframes *and* the Linux TCP/IP stack will hear it – this is a roundabout way of saying that this is equivalent to having two Ethernet switches connected to each other (except, in this virtualized case, one of them is fake/virtual).

Now, don't get too hung up on the fact that there's virtual mainframes in this picture! You could just as easily have QEMU/KVM virtual machines, simh emulators (VAX and PDPs), JNOS (packet radio stack) instances, a layer-2 VPN server/client (OpenVPN, in layer-2 mode, uses a tap device). Okay, so, what the heck is a tap device? Well, it's basically a virtual Ethernet card that you can use by `open`ing a specific file (`/dev/net/tun`), then executing an `ioctl` system call to select which tap device you want (`tap0`, `tap1234`, whatever you'd like). If you're on FreeBSD, tap devices show up like `/dev/tap0` (or whichever tap device number). When you `write` (that is, the system call) to that file, you write an array that contains a verbatim content of an Ethernet packet (minus the frame check sequence at the end of the packet – the Ethernet controller chip(s) itself will make that one for you). Now, what happens next is heavily dependent on how the tap device is connected:

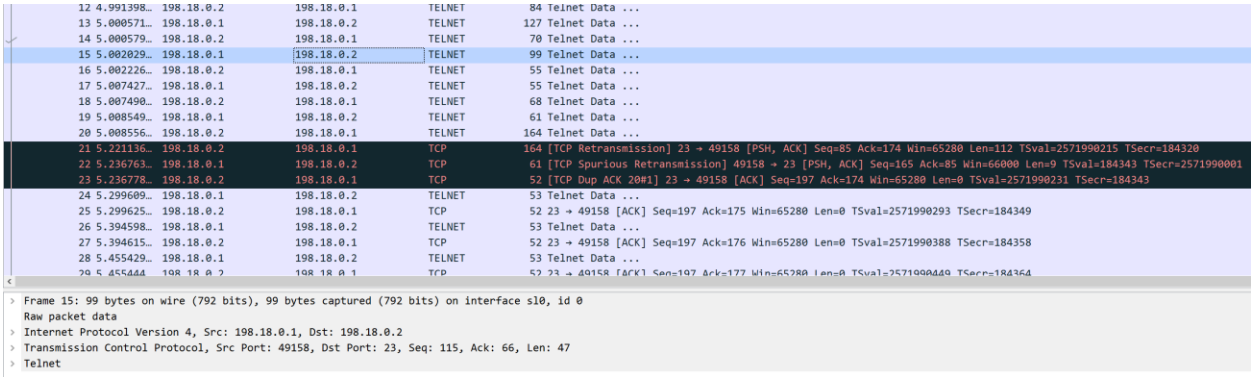
- If the tap device is associated with a bridge, the Ethernet datagram will be spit out the tap device and onto the Ethernet bridge (virtual switch). The Ethernet switch(es) will pass the datagram to the right host, by passing it along until it comes out the right port by inspecting the destination MAC address.
- If the tap device is disconnected from a bridge and it has an IP address, the Linux IP stack will see that Ethernet datagram, check if the EtherType value (which is part of something called a DIX Ethernet header and it follows the destination and source MAC address) is equal to `0x0800`, and if it is, it will try to parse it as an IP packet.
- If the tap device is neither attached to a bridge or has an assigned IP, the packet will be lost in the bit bucket of having nowhere to go.

Hercules will `open` the tap device, which already exists (hopefully; if not, Hercules invokes a program called `hercifc` and it'll make one on the spot), and that will be attached to a network interface that operates in layer-2 mode: a 3172 LCS device or OSA QDIO Ethernet running in layer-2 mode (aside the point, but, the guest OS on Hercules sets the QDIO mode – it's up to you to do something with the resultant device that shows up on the Linux host).

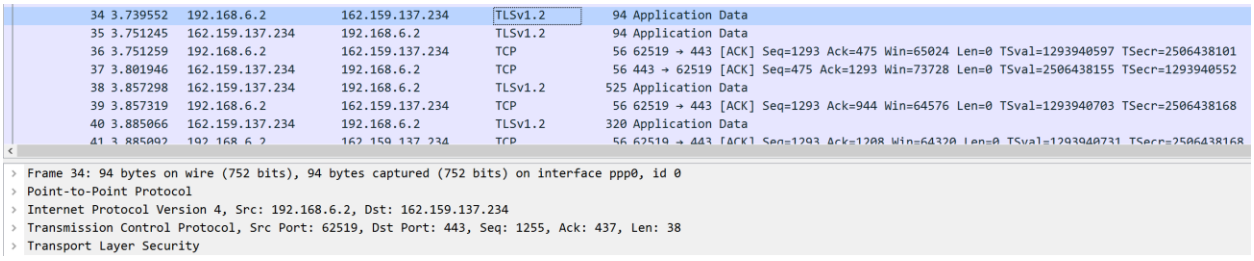
Wait, what was that thing I mentioned before about a 3088 CTCA doing TCP/IP? Well, this was what most people used on Hercules "back in the day" before the LCS emulation was perfected. Before we talk about a TCP/IP CTCA link, we have to talk about non-Ethernet TCP/IP links. If you're reading this and you're older than the authors, you probably used dial-up back in the day. What dial-up really consists of is two boxes with serial ports and telephone jacks, and, thanks to the magic of the telephone network, those two serial ports are "bridged" using phone lines. In the olden days, you'd use dial-up modems to call up a mainframe computer using a dumb terminal from a remote location. When the remote mainframe answered the call, you'd see the operating system's login prompt upon your terminal screen – you're transferring plaintext over the modem. When the internet came around, we started using dial-up modems to carry TCP/IP. In the early days of home internet connections (or, really, site-to-site internet links over serial lines and modems), we used a protocol called SLIP to get TCP/IP to flow over serial cables (and, therefore, modems). To analyze this, we're going to compare-and-contrast it against Ethernet. Let's examine a screenshot of a program called Wireshark that allows us to capture traffic on a variety of network interfaces:



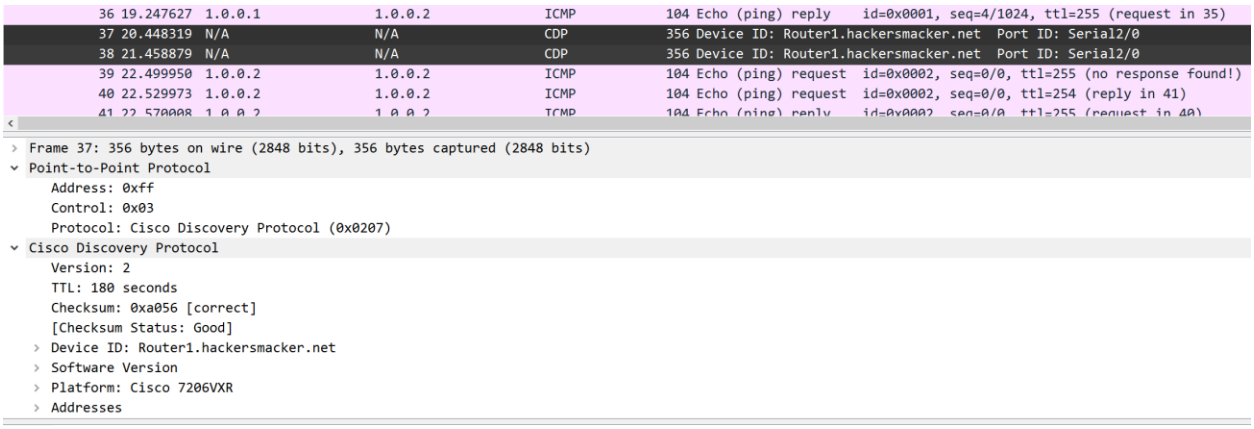
In this image, you can see some TCP packets carrying Telnet payloads. These are encapsulated into IPv4, and this is in turn encapsulated into Ethernet II (aka DIX Ethernet); this is pretty standard, this is what you'd see constantly. Because Ethernet is a link-access protocol that allows you to have more than one host attached to a single wire (though we no longer use coaxial Ethernet with the 50 ohm RF coax), it provides a simple addressing scheme using MAC addresses. The ARP protocol (a packet of which you can see just peeking over my scrollbar) maps IP addresses to MAC addresses and vice-versa. Now, let's look at a capture of a SLIP interface:



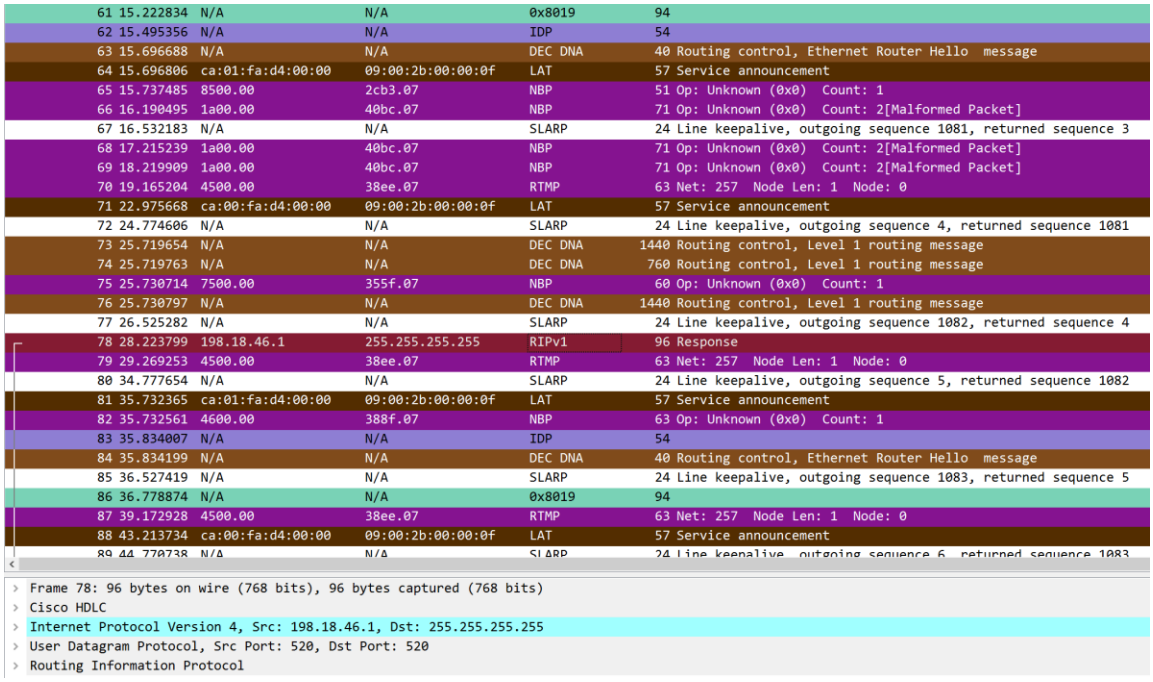
This was captured on a Linux machine with an active SLIP connection. You can see that there is *no link-layer protocol* – the IP packet is shot straight on the serial cable. In other words, there are no provisions at all for running other protocols over the interface because there's no way to know what kind of packet is going to be seen on the wire. Now, if you remember dialup, you inevitably encountered the successor to SLIP, called PPP (the Point-to-Point Protocol; in case you didn't realize, SLIP was Serial Line IP). PPP looks like this:



The PPP header itself can specify different protocols, and, a more complicated example might look like this:



CDP is a non-IP protocol – this was captured by using a serial port tee sniff adapter between two Cisco routers that were talking to each other using a regular RS-232 serial interface. In addition to PPP, there's another protocol commonly called HDLC that's actually Cisco extended HDLC (but, even then, everyone speaks it); it looks like this:

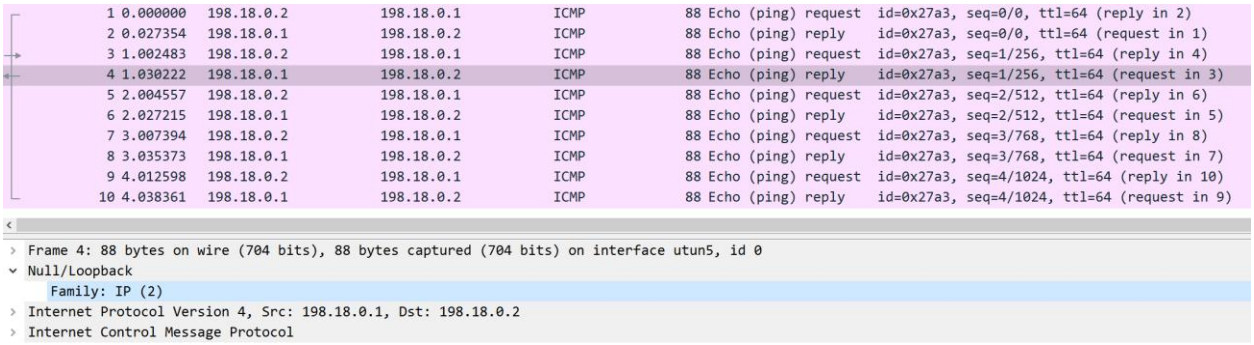


This is a true multiprotocol network attachment over a synchronous 8 megabit serial interface between two routers. This is not strictly related to mainframes, but, if you go emulate other systems (like maybe a VAX) and want to establish VPN links, you will encounter weird protocols flowing over PPP!

Okay, so, what does all of this have to do with mainframes? Well, the 3088 CTCA interface is actually a SLIP interface. Hercules emulates a *variety* of network *devices*, so, let's get those straight before we confuse ourselves:

- CTCI – Channel To Channel for IP; this links the mainframe to the host OS's TCP/IP stack by using a SLIP connection with something called a "tun device."
- CTCE – a CTC Emulation, this uses TCP sockets between two Hercules instances (possibly on the same host, possibly on a remote host). This does not create a tun or tap device.
- PTP – this is an MPCPTP link, this is a variant of CTCI that also supports IPv6. This creates a tun device.
- LCS – this is 3172 IPv4-only (unless the guest is Linux) Ethernet LCS that uses a tap device.
- QETH – this is a QDIO OSA device that might be layer-2 or layer-3. If it's layer-2, you'll get a tap device; if it's layer-3, you'll get a tun device.

So, I suppose I should tell you what a tun device is! Basically, if a tap device is a fake Ethernet card, a tun device is a fake loopback interface that functions somewhat like a mixture of PPP and SLIP. A tun can be both IPv4 and IPv6, but it cannot carry Ethernet frames. The encapsulation inside one looks like this:



It may seem a bit odd that it uses the same behavior and encapsulation as an Ethernet adapter, but, it's really quite simple! A loopback device, specifically on BSD (which the tun device works like), will accept a packet and direct it right back at the stack. A tun device, on the other hand, will allow whatever program has opened the special file that maps to the tun device to receive the packet (and transmit packets). The consequence of a tun device is that they cannot be attached to the virtual Ethernet bridge switch we have in our server, we must instead enable routing on the Linux host and either use proxy-ARP to allow the mainframe (or VPN client – tun devices are notoriously common in the VPN world) to appear on the same IP subnet as everything else, or assign them to their own IP subnet with a high netmask (such as 255.255.255.252, which provides two IPs for the mainframe and the host to use) then use routing to pass packets between the two subnets (so, enable IP forwarding on the Hercules-running Linux host).

Okay, now that you almost know enough to be dangerous, we need to now set up our virtual environment. For convenience and a lack of a desire to make people suffer, I will provide you with the commands I use. So, let's go ahead and knock it out – let's create our taps, bridges, and all of that. To experiment with networking in the lowest-difficulty possible way, I will produce a Linux guest VM running on Hercules (since running z/OS will come later). Let's prepare the networking by first surveying the landscape. Oh, I should tell you that I'm doing all of this on a custom Linux image that I've made by hand – your mileage **will** vary on other distributions! Okay, let's see what we've got:

```
# uname -a
Linux linux-pcvm 6.5.2 #17 SMP PREEMPT_DYNAMIC Mon Nov 11 08:25:32 CST 2024 x86_64 GNU/Linux
# ifconfig eth0
Eth0: flags=4098<BROADCAST,MULTICAST>  mtu 1500
        ether 08:00:27:79:94:5d  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
```

```
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

#
```

Okay, so, we've got ourselves an Ethernet card! Looks like the card's deactivated right now, so, let's go ahead and get it ready:

```
# brctl addbr br0
# tunctl -t tap0
Set 'tap0' persistent and owned by uid 0
# brctl addif br0 tap0
# brctl addif br0 eth0
# ifconfig eth0 up
# ifconfig br0 192.168.1.122 up
# brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.5a684326c9b4       no              eth0
                                                        tap0
#
```

So, first, I created the virtual Ethernet switch. Then, I created a tap device for Hercules to use on an LCS device. Next, I attached both the tap and eth1 card to the bridge, upped the interfaces, and gave the bridge an IP address. In this case, Linux will be known to the outside network as 192.168.1.122 – this is, I know, different from my diagram above. Let's review the progress so far:

```
# ifconfig
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.122  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fd12:3456:7890:1234:901f:a6ff:fe04:1acb  prefixlen 64  scopeid
0x0<global>
        inet6 fe80::901f:a6ff:fe04:1acb  prefixlen 64  scopeid 0x20<link>
        inet6 fd12:3456:7890:1234:738b:72f8:93f9:ad91  prefixlen 64  scopeid
0x0<global>
        ether 92:1f:a6:04:1a:cb  txqueuelen 1000  (Ethernet)
        RX packets 1813  bytes 345792 (345.7 KB)
        RX errors 0  dropped 52  overruns 0  frame 0
        TX packets 66  bytes 10631 (10.6 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::a00:27ff:fe6c:e465  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:6c:e4:65  txqueuelen 1000  (Ethernet)
        RX packets 108857  bytes 148345931 (148.3 MB)
        RX errors 0  dropped 5593  overruns 0  frame 0
        TX packets 13469  bytes 1065046 (1.0 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
```

```
        RX packets 146  bytes 14443 (14.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 146  bytes 14443 (14.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

tap0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether 8e:30:dc:04:2c:82  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

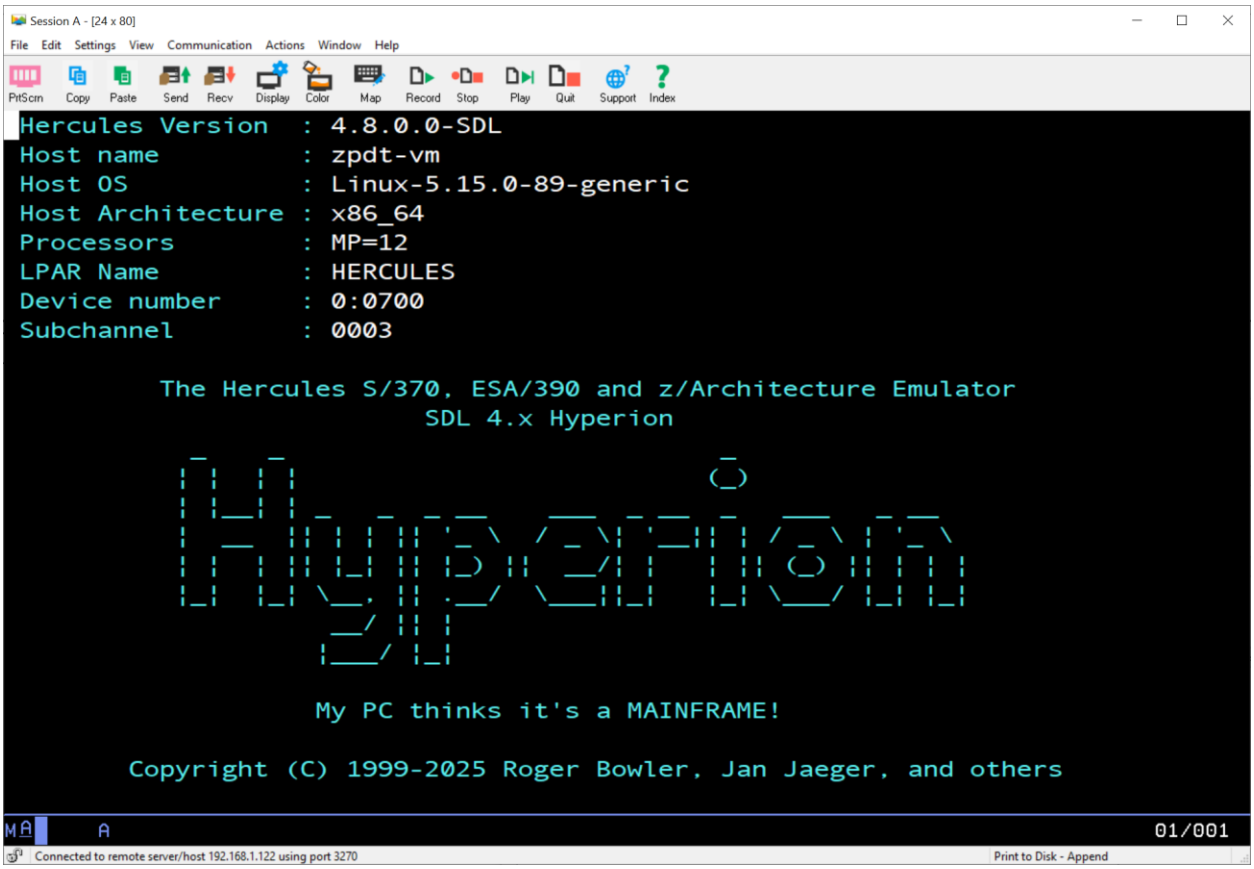
Okay, looks like we're good. The bridge interface itself has IPv4 and IPv6 – and, it all works! Now, we can get into the meat and potatoes: making Hercules work. Once you've compiled and installed the latest version of Hyperion, proceed onwards.

Making Guests Run

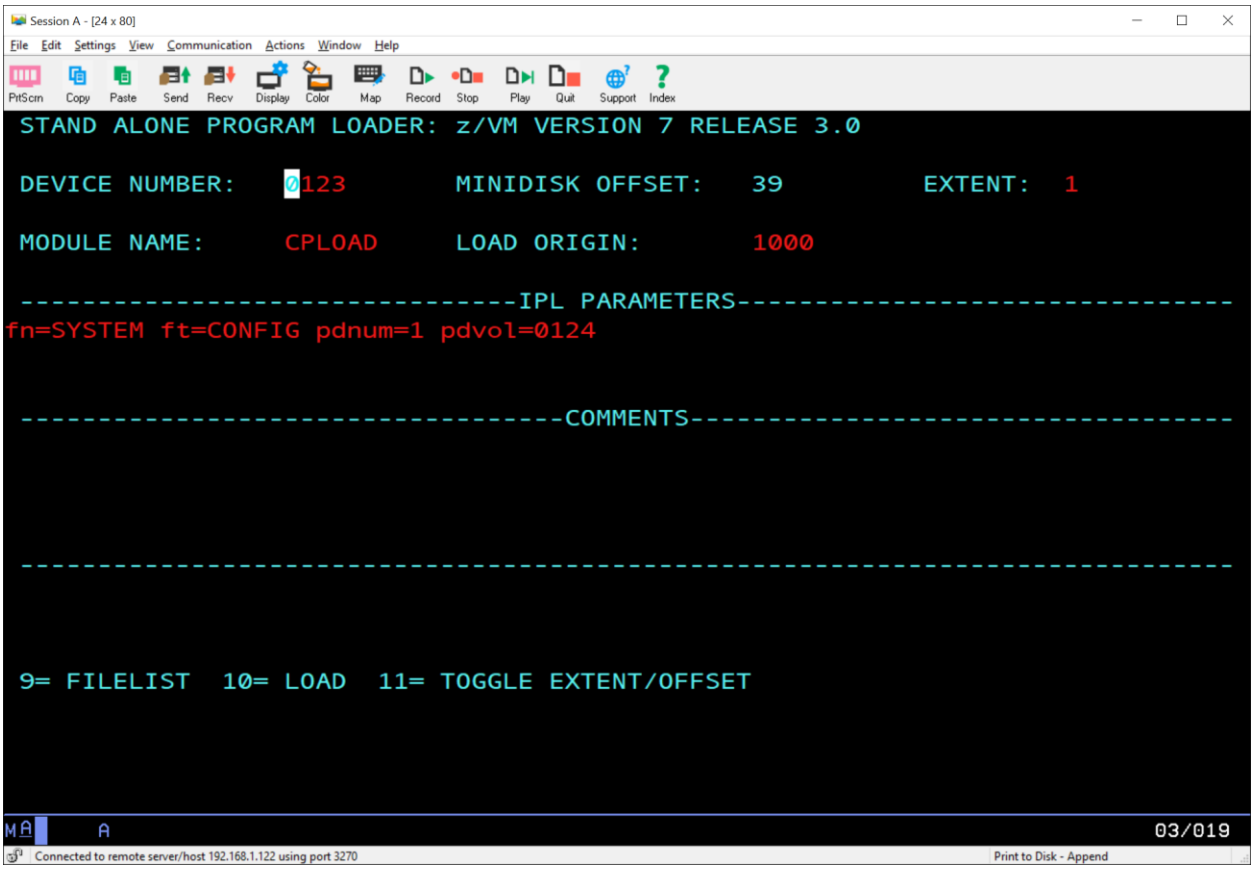
In this very simple exercise, we will "install" (run a premade system) the simplest guest: z/VM. Later on, we're going to configure z/VM and make our z/Linux guest run under it. Now, I'm going to use z/VM 7.3, which is attainable from the IBM student software catalog if you dig a bit. Otherwise, just imagine it as a stand-in for any z/VM version – they're essentially all the same since VM/ESA 1.2. Fire up an editor, and start writing a Hercules config file, `hercules.cnf`, in some working directory:

```
CPUSERIAL   000777
CPUMODEL    2818
MAINSIZE    8192
CNSLPORT    3270
ARCHLVL     Z/ARCH
FACILITY    ENABLE    BIT44
FACILITY    QUERY    BIT44
MAXCPU      8
NUMCPU      8
LOADPARM    0700.... ← note: if you leave this off or blank (all dots), you will
use Hercules's console as z/VM's console as this emulates the HMC integrated
console
DIAG8CMD    ENABLE
PGMPRDOS    LICENSED
OSTAILOR    QUIET
LEGACYSENSEID    ENABLE

000C    3505    3555 sockdev ascii trunc eof
000D    3525    pun00d.txt ascii crlf
000E    1403    prt00e.txt crlf

0700.4  3270

0123    3390    M01RES cu=3990-6
0124    3390    VMCOM1 cu=3990-6
0125    3390    730RL1 cu=3990-6
0126    3390    M01S01 cu=3990-6
0127    3390    M01P01 cu=3990-6
```

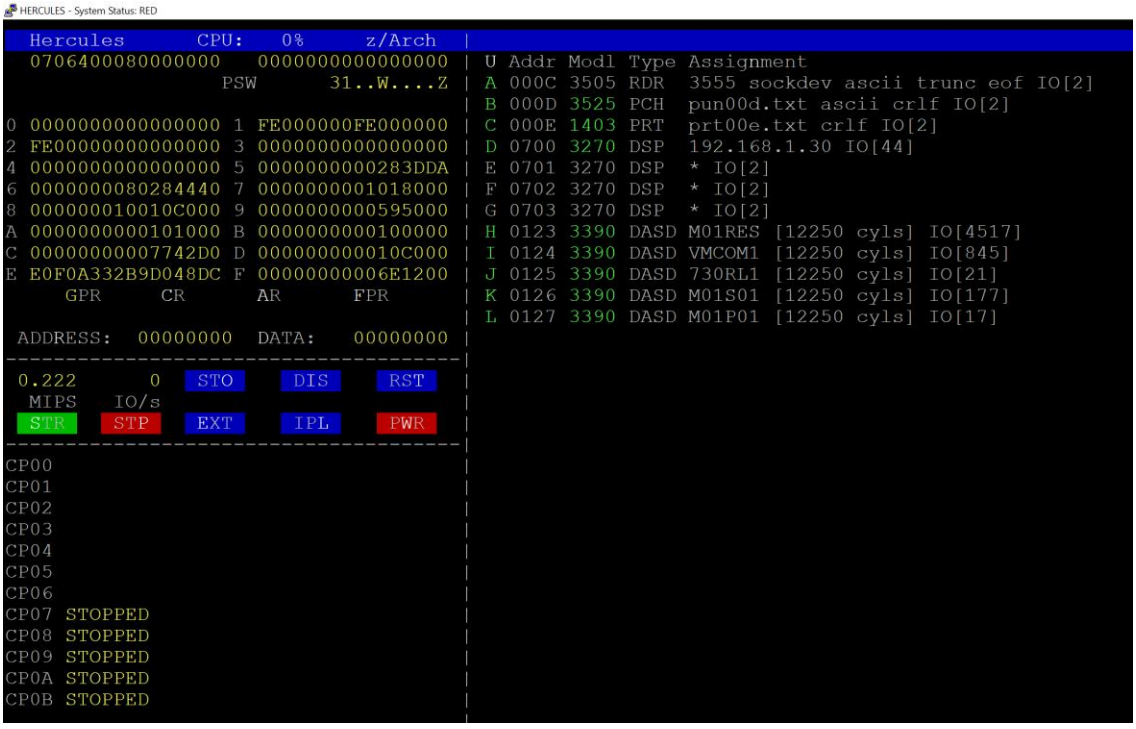
Go ahead and run Hercules from the directory where everything is, and IPL with `ipl 123` after you connect a 3270 console:



Then, IPL and strike PF10 to bring up VM. This guide is not a tutorial on VM at all – just a guide on getting Hercules to work the best. Here's the IPL screen, on the next page:



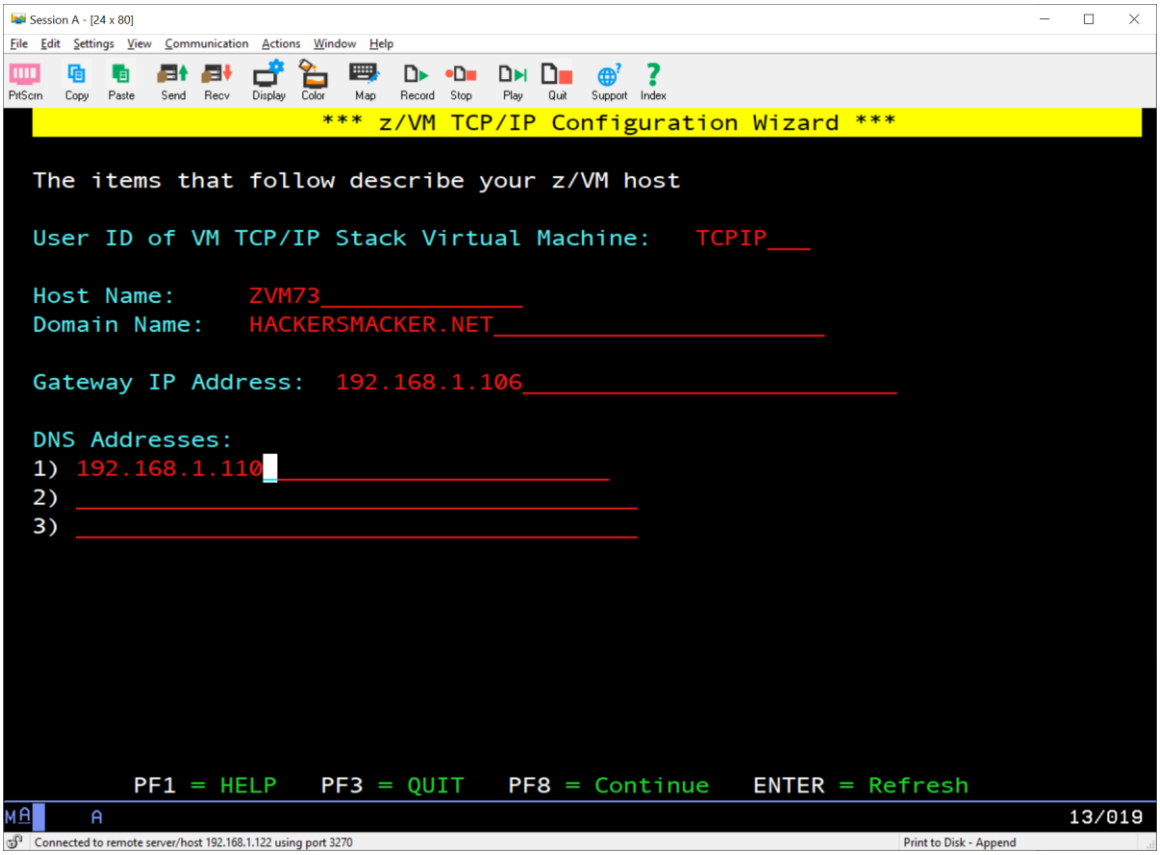
Strike ESC on the Hercules console to see the specs:

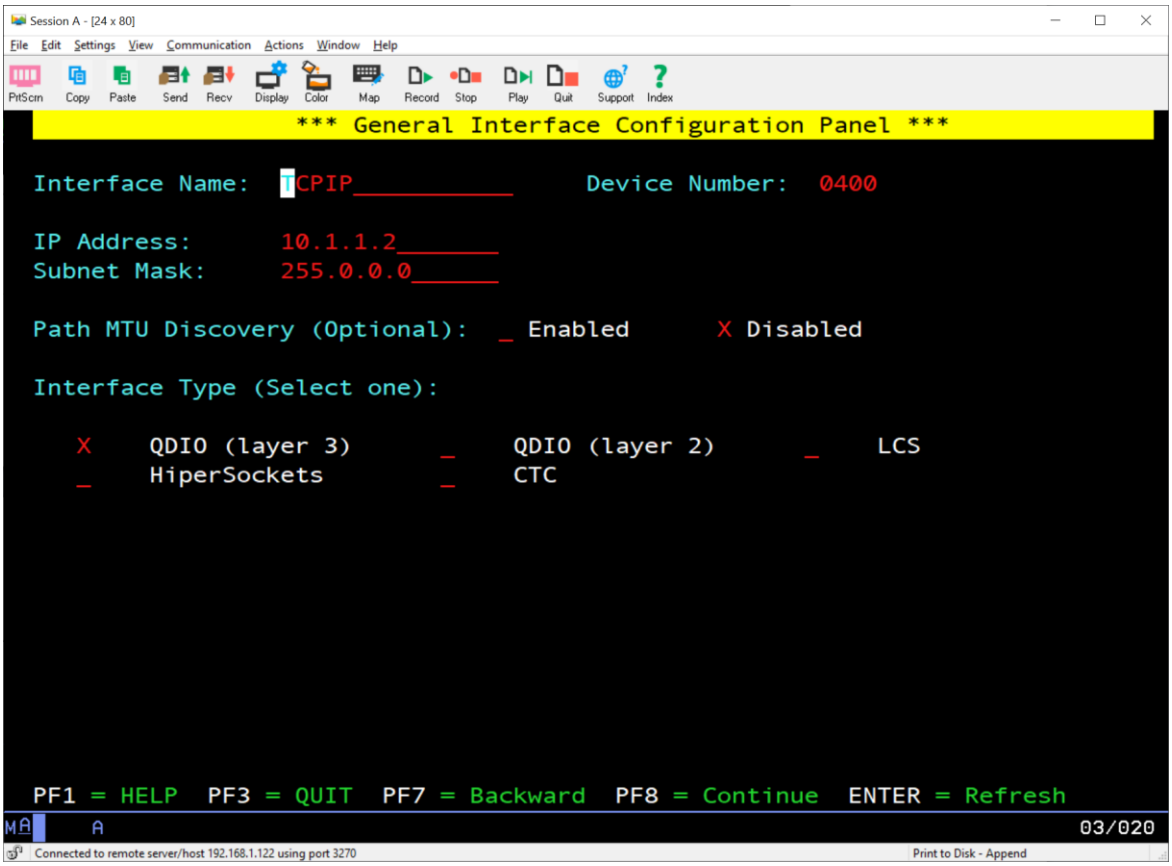


You can see that z/VM's nucleus is idling some of the CPCs (Central Processor Complexes) – this is normal. Once load increases, the extra CPCs will start. On the z/VM 7.3 image, the default login is maint730 with a password of zvm730, so, log in, and figure out what the password for the TCP/IP service machine userid is by reading the directory, stored in `USER DIRECT C1`:

```
IDENTITY TCPIP    LBYONLY   128M  256M ABG
 INCLUDE TCPCMSU
 BUILD ON * USING SUBCONFIG TCPIP-1
* BUILD ON @@member2name USING SUBCONFIG TCPIP-2
* BUILD ON @@member3name USING SUBCONFIG TCPIP-3
* BUILD ON @@member4name USING SUBCONFIG TCPIP-4
 ACCOUNT IBM
 LOGONBY IBMVM1
 OPTION QUICKDSP SVMSTAT MAXCONN 1024 DIAG98 APPLMON
 SHARE RELATIVE 3000
 IUCV ALLOW
 IUCV ANY PRIORITY
 IUCV *CCS PRIORITY MSGLIMIT 255
 IUCV *VSWITCH MSGLIMIT 65535
```

Looks like we need to set a password for this userid! Replace LBYONLY with ZVM730, then run `DIRECTXA USER DIRECT C` to burn the directory to the system. This isn't super necessary, but it does allow us to stare at the TCP/IP console without doing the LOGON BY mess. Okay, as MAINT, access disk 193 as mode D and invoke IPWIZARD:





Well well well, look at that! There's our network choices – layer-2 or layer-3 QDIO OSA, an LCS 3172, HiperSockets (ignore that one, that's for inter-z/VM communication between multiple LPARs), and a CTC. Plug something reasonable in here. Before proceeding, go over to the Hercules operator console and enter a command to create a QETH card for our tap device:

```
HHC01603I attach 0600.3 qeth ifname tap0
14:33:25 HCPMCI9101I MACHINE CHECK ON CPU 0003.  MCIC = 00400F1D 403B0000
14:33:25 HCPMCI9109I System operation continues.
14:33:25 HCPMCI9101I MACHINE CHECK ON CPU 0001.  MCIC = 00400F1D 403B0000
14:33:25 HCPMCI9109I System operation continues.
14:33:25 HCPMCI9101I MACHINE CHECK ON CPU 0004.  MCIC = 00400F1D 403B0000
14:33:25 HCPMCI9109I System operation continues.
14:33:25 HCPRFC2264I Device 0600 is available and online.
14:33:25 HCPRFC2264I Device 0601 is available and online.
14:33:25 HCPRFC2264I Device 0602 is available and online.
```

Note that there is output here that is bolded – that's from z/VM's nucleus (CP) telling us the device suddenly appeared and it wasn't expected. That's fine. Okay, go enter everything on that IPWIZARD dialog, and kick off TCP/IP with `XAUTOLOG TCPIP`. Hopefully it works:

```
14:36:49 AUTO LOGON  ***       TCPIP    USERS = 14    BY MAINT730
14:36:57 USER DSC    LOGOFF AS  TCPIP    USERS = 13    FORCED BY MAINT730
14:37:00 AUTO LOGON  ***       TCPIP    USERS = 14    BY MAINT730
14:37:01 0600-0602 ATTACHED TO TCPIP BY TCPIP
```

```
HHC00809I Processor CP0A: disabled wait state 0102400080000000 0000000000000000
HHC00826W Processor CP0A: processor auto-stopped due to disabled wait
HHC00007I Previous message from function 'z900_process_interrupt' at cpu.c(1883)
HHC00809I Processor CP09: disabled wait state 0102400080000000 0000000000000000
HHC00826W Processor CP09: processor auto-stopped due to disabled wait
HHC00007I Previous message from function 'z900_process_interrupt' at cpu.c(1883)
14:37:05 USER DSC   LOGOFF AS  TCPIP    USERS = 13    FORCED BY MAINT730
14:37:05 OSA   0600 DETACHED TCPIP    0600 BY TCPIP
14:37:05 OSA   0601 DETACHED TCPIP    0601 BY TCPIP
14:37:05 OSA   0602 DETACHED TCPIP    0602 BY TCPIP
14:37:08 AUTO LOGON  ***       TCPIP    USERS = 14    BY MAINT730
14:37:09 0600-0602 ATTACHED TO TCPIP BY TCPIP
HHC03800I 0:0602 qeth: Adapter mode set to Layer 2
HHC00901I 0:0602 qeth: Interface tap0, type TAP opened
HHC03997I 0:0602 qeth: tap0: using MAC address 8E:30:DC:04:2C:82
HHC03997I 0:0602 qeth: tap0: using MTU 1500
HHC02499I Hercules utility hercifc - Hercules Network Interface Configuration
Program - version 4.8.0.0-SDL
HHC01414I (C) Copyright 1999-2025 by Roger Bowler, Jan Jaeger, and others
HHC01417I ** The SDL 4.x Hyperion version of Hercules **
HHC01415I Build date: May 25 2025 at 17:46:18
HHC03801I 0:0602 qeth: tap0: Register guest MAC address 02:00:00:00:00:01
HHC03801I 0:0602 qeth: tap0: Register guest MAC address 01:00:5e:00:00:01
```

So, it bounced once then it properly came up. That hercifc program got called upon, and it attached to the tap device. Go back over to your CMS session and run some commands to see that it all works good:

```
vmlink tcpmaint 592
DMSVML2060I TCPMAINT 592 linked as 0120 file mode Z
Ready; T=0.03/0.04 14:38:29
ping 192.168.1.122
Ping Level 730: Pinging host 192.168.1.122.
                Enter #CP EXT to interrupt.
PING: Ping #1 response took 0.006 seconds. Successes so far 1.
Ready; T=0.16/0.51 14:38:51
ping 192.168.1.1
Ping Level 730: Pinging host 192.168.1.1.
                Enter #CP EXT to interrupt.
PING: Ping #1 response took 0.006 seconds. Successes so far 1.
Ready; T=0.16/0.50 14:38:55
```

Looks good! We can ping the host on its bridge interface Linux TCP/IP stack, and we can also ping hosts outside of the virtual switch. This works perfectly! Now, log in as TCPMAINT (you'll need to set its password in the same way you did TCPIP's), and go look for a file named PROFILE TCPIP. Open that with XEDIT, and obliterate whatever's in there and replace it with this:

```
OBEY
OPERATOR TCPMAINT MAINT MPROUTE REXECD SNMPD SNMPQE LDAPSRV MAINT730
ENDOBEY
; ------------------------------------------------------------------
PORT
  20   TCP FTPSERVE  NOAUTOLOG ; FTP Server
  21   TCP FTPSERVE            ; FTP Server
```

```
  23   TCP INTCLIEN            ; TELNET Server
  25   TCP SMTP                ; SMTP Server
  111  TCP PORTMAP             ; Portmap Server
  111  UDP PORTMAP             ; Portmap Server
  161  UDP SNMPD               ; SNMP Agent
  162  UDP SNMPQE              ; SNMPQE Agent
; 389  TCP LDAPSRV             ; LDAP Server
; 389  UDP LDAPSRV             ; LDAP Server
  512  TCP REXECD              ; REXECD Server (REXEC)
  514  TCP REXECD              ; REXECD Server (RSH)
  520  UDP MPROUTE   NOAUTOLOG ; Multiple Protocol Routing Server
; 608  TCP UFTD                ; UFT Server
; 636  TCP LDAPSRV             ; LDAP Server (Secure)
; 636  UDP LDAPSRV             ; LDAP Server (Secure)
  2049 UDP VMNFS               ; NFS Server
  2049 TCP VMNFS     NOAUTOLOG ; NFS Server
; ------------------------------------------------------------------------
DEVICE DEV@0600  OSD 0600
LINK ETH0 QDIOETHERNET DEV@0600 MTU 1500 ETHERNET ENABLEIPV6
; ------------------------------------------------------------------------
HOME
192.168.1.123 255.255.255.0 ETH0
FD12:3456:7890:1234::123 ETH0
; ------------------------------------------------------------------------
GATEWAY
; Network       Subnet          First           Link            MTU
; Address       Mask            Hop             Name            Size
; ------------ --------------- --------------- ---------------- -----
DEFAULTNET                     192.168.1.106   ETH0            1500
198.18.73.1     255.255.255.255 =               ETH1            1500
; ------------------------------------------------------------------------
START DEV@0600
```

This is more ideal because this does IPv6 – also, I enabled path MTU discovery (which is the implicit default). This is working absolutely swimmingly in layer-2 mode! Now, let's max it out. Let's add us a layer-3 device:

```
HHC01603I attach 0603.3 qeth
14:51:26 HCPMCI9101I MACHINE CHECK ON CPU 0004.  MCIC = 00400F1D 403B0000
14:51:26 HCPMCI9109I System operation continues.
14:51:26 HCPMCI9101I MACHINE CHECK ON CPU 0004.  MCIC = 00400F1D 403B0000
14:51:26 HCPMCI9109I System operation continues.
14:51:26 HCPMCI9101I MACHINE CHECK ON CPU 0009.  MCIC = 00400F1D 403B0000
14:51:26 HCPMCI9109I System operation continues.
14:51:26 HCPRFC2264I Device 0603 is available and online.
14:51:26 HCPRFC2264I Device 0604 is available and online.
14:51:26 HCPRFC2264I Device 0605 is available and online.
```

Note that I didn't specify any interface names – since we can't use a tunnel device on a bridge and Linux will automatically detect the point-to-point nature of the interface and install routes, it doesn't particularly matter. Okay, let's get that configured. First, go edit `SYSTEM DTCPARMS D` as TCPMAINT and go update the device address range to attach. I hope you know how to configure z/VM TCP/IP – it's super easy – enter these options:

```
DEVICE DEV@0603  OSD 0603
```

```
LINK ETH1 QDIOETHERNET DEV@0603 IP ENABLEIPV6
HOME
198.18.73.2 255.255.255.252 ETH1
START DEV@0603
```

Cycle TCP/IP. On the host, run this commands to make that interface usable:

```
# ifconfig tun0 198.18.73.1 pointtopoint 198.18.73.2
```

This tells the Linux kernel it can access the z/VM system through this interface at that destination address, and the Linux TCP/IP stack uses 198.18.73.1 as its own IP on that subnet. Because the device is a point-to-point tunnel interface, the effective requirement here is that Linux must now be a router. No matter, go enable forwarding and you should be good to go. Alas, this isn't too terribly useful in this context – we already have a QDIO Ethernet layer-2 device sitting on the right LAN to not necessitate routing.

Here's the device entry for a regular LCS and also a CTC:

```
HHC01603I attach 0e20.2 lcs tap1
HHC00901I 0:0E20 lcs: Interface tap1, type TAP opened
HHC00942I CTC: lcs interface tap1 using mac 7A:06:F9:58:62:80
HHC00921I CTC: lcs device port 00: manual Multicast assist enabled
HHC00935I CTC: lcs device port 00: manual Checksum Offload enabled
15:25:41 HCPMCI9101I MACHINE CHECK ON CPU 0001.  MCIC = 00400F1D 403B0000
15:25:41 HCPMCI9109I System operation continues.
15:25:41 HCPMCI9101I MACHINE CHECK ON CPU 0002.  MCIC = 00400F1D 403B0000
15:25:41 HCPMCI9109I System operation continues.
15:25:41 HCPRFC2264I Device 0E20 is available and online.
15:25:41 HCPRFC2264I Device 0E21 is available and online.
HHC01603I attach 0e22.2 ctci 198.18.74.2 198.18.74.1
HHC00901I 0:0E22 ctci: Interface tun1, type TUN opened
15:26:27 HCPMCI9101I MACHINE CHECK ON CPU 0005.  MCIC = 00400F1D 403B0000
15:26:27 HCPMCI9109I System operation continues.
15:26:27 HCPMCI9101I MACHINE CHECK ON CPU 0008.  MCIC = 00400F1D 403B0000
15:26:27 HCPMCI9109I System operation continues.
15:26:27 HCPRFC2264I Device 0E22 is available and online.
15:26:27 HCPRFC2264I Device 0E23 is available and online.
HHC02499I Hercules utility hercifc - Hercules Network Interface Configuration
Program - version 4.8.0.0-SDL
HHC01414I (C) Copyright 1999-2025 by Roger Bowler, Jan Jaeger, and others
HHC01417I ** The SDL 4.x Hyperion version of Hercules **
HHC01415I Build date: May 25 2025 at 17:46:18
```

Note that I specified the source and destination IP directly on the attachment command that bought in the CTCI device. You cannot do this with a layer-3 OSA – only a CTCI. Also, this uses tap1 for the LCS – you need to create that and bind it to the bridge just like seen before. Enter this into the PROFILE TCPIP file:

```
DEVICE DEV@0E20 LCS 0E20
LINK ETH2 ETHERNET 0 DEV@0E20
DEVICE DEV@0E22 CTC 0E22
LINK ETH3 CTC 0 DEV@0E22
HOME
192.168.1.125 255.255.255.0 ETH2
```

```
198.18.74.2 255.255.255.252 ETH3
GATEWAY
198.18.74.1     255.255.255.255 =               ETH3            1500
START DEV@0E20
START DEV@0E22
```

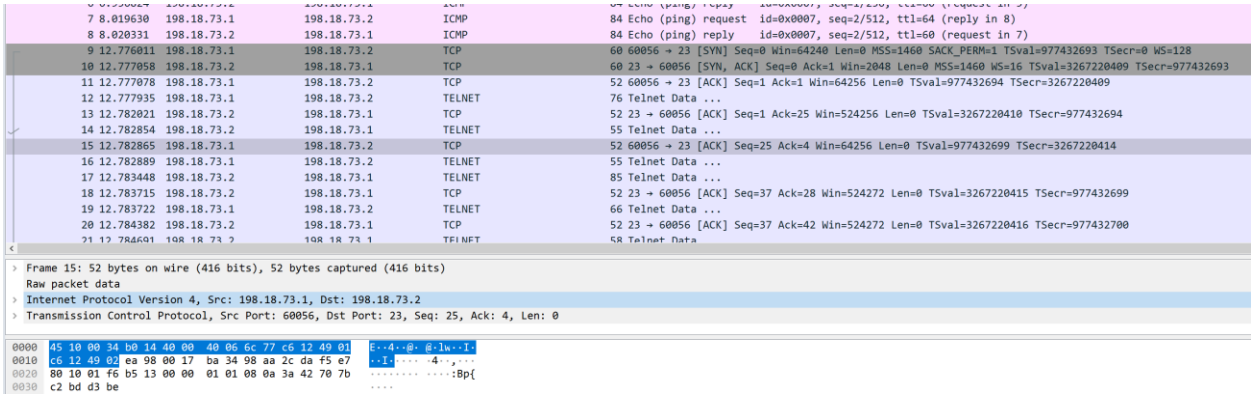
Okay, there we go. Restart TCP/IP, and, hopefully… watch it fly!

```
M 0600 1731 OSA  QDIO tap0 tx[140] rx[57040] IO[72]
N 0601 1731 OSA  QDIO IDX IO[173]
O 0602 1731 OSA  QDIO IDX IO[165]
P 0603 1731 OSA  QDIO tun0 tx[66] rx[52] IO[54]
Q 0604 1731 OSA  QDIO IDX IO[154]
R 0605 1731 OSA  QDIO IDX IO[150]
S 0E20 3088 CTCA LCS Port 00 IP Pri (tap1) IO[425]
T 0E21 3088 CTCA LCS Port 00 IP Pri (tap1) IO[20]
U 0E22 3088 CTCA CTCI 198.18.74.2/198.18.74.1 (tun1) IO[18]
V 0E23 3088 CTCA CTCI 198.18.74.2/198.18.74.1 (tun1) IO[14]
```

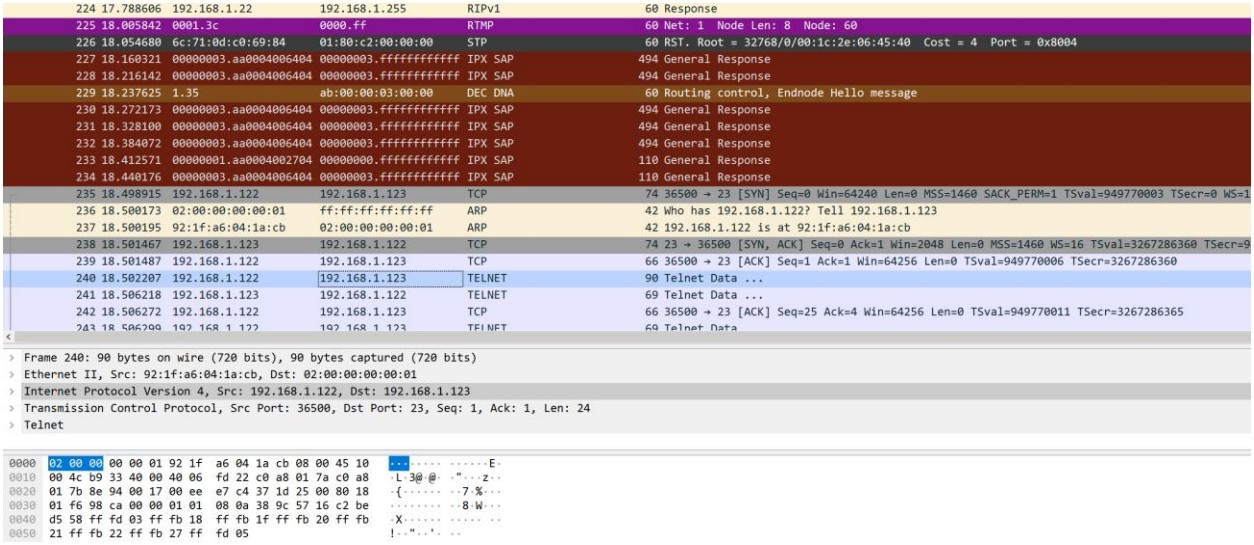
You can see the resultant interfaces that get created on the host, too – they work great!

I won't explain how to set up SNA in great detail, but you do the same thing as the normal LCS, but you run `attach 0E22 LCS -e SNA tap2` to attach it to Hercules (yes, it is a single-ended device). For information on configuring SNA, go consult the HSnet web page and read the SNA tutorial present on the site – the site's author has made it relatively simple!

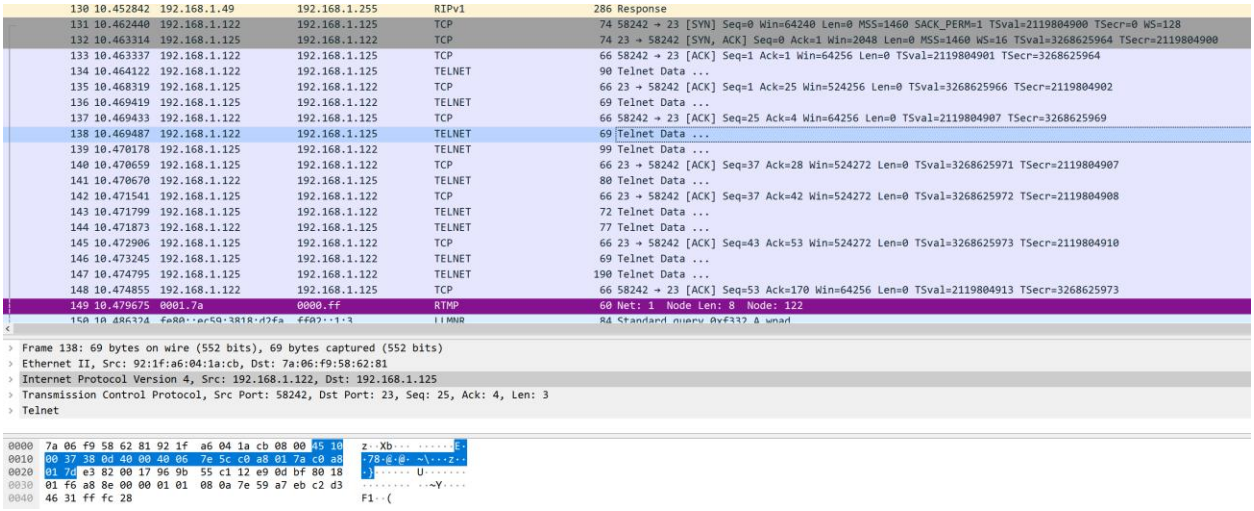
Just to cover down on the various operating modes of the device, let's examine some Wireshark captures on the virtframe side of the various cards. First, let's examine a layer-3 QDIO Ethernet card (device name tun0):



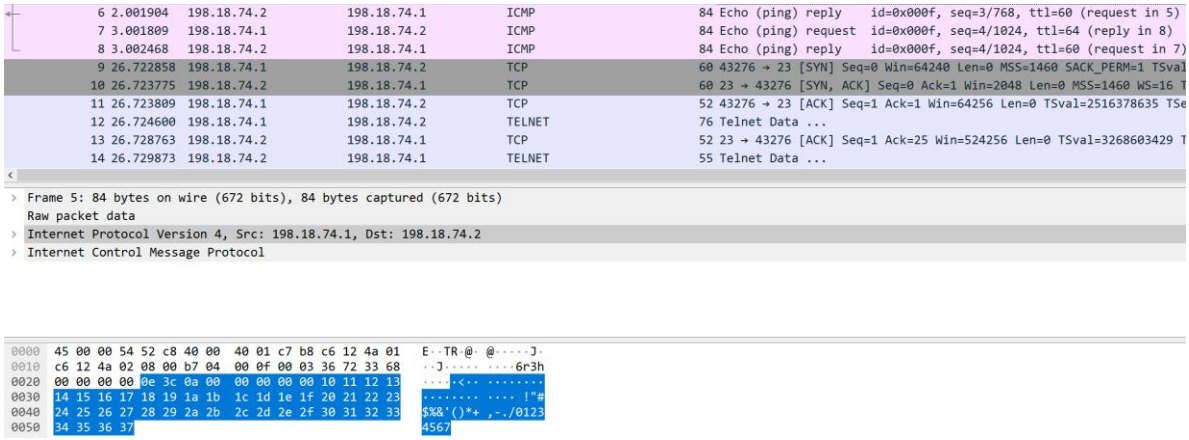
Now, QDIO layer-2 Ethernet. Before looking at this next image, please do note that I have connected the virtframe's resultant tap0 device that got created onto a bridge as shown in this article, and is attached therefore to my server LAN. The downstream guest that is hooked into the device will parse any resultant Ethernet frames that come in – this is necessary for the operation of a z/VM VSWITCH (which can use a layer-2 QDIO Ethernet as a "trunk port", possibly with 802.1Q VLAN tagging enabled). The image shows on the next page.



An LCS interface looks similar, but ONLY IPv4 and ARP packets will make it "through" to the VM (unlike a layer-3 QDIO Ethernet, where *everything* fits through):



Finally, a CTCI IPv4-only interface:



Hercules disks

DASDs, as they're called on mainframes, come in a variety of shapes and sizes. There's CKD DASDs, which are the standard variable-sector-size DASDs you need to run OSes of the MVS line (up to and including z/OS). The opposite of CKD is FBA, and these are basically normal disks with standard 512-byte sectors. Since CKD DASDs inevitably incur a performance overhead, using FBA DASDs is **very** advisable where applicable. When installing z/VM or z/VSE, *install it onto FBA DASDs. You will thank me later when the performance soars.*

So, then, what are the most common DASDs you'll see?

Model	Type	Size
3390-3	CKD	3339 cyls, 946 MB
3390-9	CKD	10017 cyls, 8.51 GB
3380-K	CKD	2655 cyls, 1.890 GB
9336-20	FBA	Usually any
3370	FBA	Usually any

FBA DASDs tend to be any size and only use that model number as a stand-in for a real disk device. Hercules, as such, can use real disks as FBA DASDs! You can partition a real disk into extents and attach those to Hercules, or you can just give it the entire disk. Oh, CKD DASDs… those still have to be emulated as files. If I were you, I'd store them uncompressed (do not specify the –z option on the dasdinit program) and as single files on the lowest-overhead filesystem you can find. I recommend using XFS or JFS on Linux for this very reason.

I have prepared a blank disk at `/dev/sdb`, let us attach it to Hercules:

```
HHC01603I attach 200 3370 /dev/sdb
HHC00504I 0:0200 FBA file /dev/sdb: REAL FBA opened
HHC00507I 0:0200 FBA file /dev/sdb: origin 0, blks 20971520
16:03:13 HCPMCI9101I MACHINE CHECK ON CPU 0001.  MCIC = 00400F1D 403B0000
16:03:13 HCPMCI9109I System operation continues.
16:03:13 HCPRFC2264I Device 0200 is available and online.
```

This sure looks good, we're using a REAL FBA… for a REAL USER! Okay, let's write us up a Linux directory entry:

```
USER LINUX1 ZVM730 2G 2G BG
 MACHINE ESA 8
 CPU 00
 CPU 01
 CPU 02
 CPU 03
 CPU 04
 CPU 05
 CPU 06
 CPU 07
 OPTION MAINTCCW RMCHINFO
 CONSOLE 009 3215 T OPERATOR
 DEDICATE 200 200
 DEDICATE 600 606
 DEDICATE 601 607
 DEDICATE 602 608
 LINK MAINT 190 190 RR
```

```
 LINK MAINT 19D 19D RR
 LINK MAINT 19E 19E RR
 SPOOL 00C 3505
 SPOOL 00D 3525
 SPOOL 00E 3211 A
 SPECIAL 300 3270
 SPECIAL 301 3270
```

If you have working DIRMAINT, load it in (again, this guide is a performance optimization guide, not a z/VM tutorial).

Get the kernel files transferred:

```
vmlink tcpmaint 592
ftp 198.18.74.1
VM TCP/IP FTP Level 730
Connecting to 198.18.74.1, port 21
220 ProFTPD Server (Debian) Ý::ffff:198.18.74.1¨
USER (identify yourself to the host):
user
>>>USER user
331 Password required for user
Password:
password
>>>PASS ********
230 User user logged in
Command:
cd /cdrom/boot
Command:
bin
Command:
locsite fix 80
Command:
get linux_vm debian.linux
Command:
get root.bin debian.initrd
Command:
ascii
Command:
get parmfile debian.parmfile
Command:
quit
>>>QUIT
221 Goodbye.
Ready; T=0.32/0.76 16:20:04
```

Okay, edit DEBIAN EXEC to use the right files and target VMs:

```
/* REXX LOAD EXEC FOR DEBIAN LINUX S/390 VM GUESTS */
'CP CLOSE RDR'
'SPOOL PUNCH LINUX1'
'PUNCH DEBIAN LINUX A (NOH'
'PUNCH DEBIAN PARMFILE A (NOH'
'PUNCH DEBIAN INITRD A (NOH'
```

Run the EXEC:

debian

```
PUN FILE 0013 SENT TO   LINUX1   RDR AS  0001 RECS 086K CPY  001 A NOHOLD NOKEEP
PUN FILE 0014 SENT TO   LINUX1   RDR AS  0002 RECS 0003 CPY  001 A NOHOLD NOKEEP
PUN FILE 0015 SENT TO   LINUX1   RDR AS  0003 RECS 167K CPY  001 A NOHOLD NOKEEP
Ready; T=0.67/10.62 16:25:27
```

Go log in as that user, and `IPL 00C`.

```
RDR FILE 0001 SENT FROM MAINT730 PUN WAS 0013 RECS 086K CPY  001 A NOHOLD NOKEEP
RDR FILE 0002 SENT FROM MAINT730 PUN WAS 0014 RECS 0003 CPY  001 A NOHOLD NOKEEP
RDR FILE 0003 SENT FROM MAINT730 PUN WAS 0015 RECS 167K CPY  001 A NOHOLD NOKEEP
00:
00: CP
00: ipl 00c clear
00:      NO FILES CHANGED
<KERNEL OUTPUT>
```

When it actually comes to doing the installation, it's pretty easy. Configure the QDIO card you've got attached, and SSH into the system as the installer user. Now, since we're using FBA, we have to do something important. Go through the installer up to the partitioning part, then press control-A then N to go to the next window on Screen (in case you didn't know, the bar at the top is from GNU Screen). In there, format the target DASD:

```
# mkfs.xfs /dev/dasda1
```

Switch back to the installer, and go through the partitioner. Use that as the "XFS journaling filesystem", position its mount at /, finish, and you'll see a dialog about not needing to write any partition table changes – this is to be expected. Then, proceed with the installer. This is going to take a while, so, sit back and enjoy the ride!

Many-core versus fast-core

In the course of mainframe emulation, a common question has often come to many minds: what is superior? A dual-socket server with two Epycs (rounding up to 256 cores), or a screaming-fast Ryzen clocking at 5.8 GHz? Well, the common answer by some is that the Epyc machine will be superior, but, this is not always the case. Hercules (as well as simh, AlphaVM, QEMU, or any emulator for the most part) will fare much better with a single-processor system. Generally speaking, you can generally score a better "user experience" by selecting a modern Ryzen 9 9xxx chip – these are very overclockable, and feature a decent bit of cache memory (important for emulators – the more, the better).

Many people make the fatal mistake of trying to build out their virtframes on a 10 year old Dell PowerEdge with a 2.8 GHz CPU and slow DDR3 RAM – don't be this person! Single-threaded performance is what you want: select for it as it also means better single-throughput I/O performance (which is ideal for SSDs where raw speed matters).