ARL-TN-1278 ● SEP 2025

Intra-host IP Networking in Support of Simulation

by Trevor Cook

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TN-1278 ● SEP 2025

Intra-host IP Networking in Support of Simulation

Trevor Cook
DEVCOM Army Research Laboratory

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
September 2025	Technical Note	10/1/2024	9/3/2025

4. TITLE AND SUBTITLE
Intra-host IP Networking in Support of Simulation

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**

6. AUTHOR(S)
Trevor Cook

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
DEVCOM Army Research Laboratory ATTN: FCDD-RLA-NB Adelphi, MD 20783	ARL-TN-1278

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report details a method of emulating an entire Extendable Mobile Ad-hoc Network Emulator (EMANE) network on a single host operating system. The solution uses a novel IP addressing scheme, network address translation, and Linux kernel routing rules to bounce packets between a single host's network interfaces as if they were traversing an actual network. The system was designed to enable the integration of a numerical simulation with military communications, and the implementation details illustrate low-level Linux routing techniques.

15. SUBJECT TERMS
Networking, Cyber and Computational Sciences; Network Address Translation; Linux; Network Emulation; Extendable Mobile Ad-hoc Network Emulator; EMANE

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UU	25

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Trevor Cook	(520) 691-5685

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

1. Introduction

I was recently faced with the challenge of adding a degree of military relevance to a network simulation that estimated key generation rates produced by a quantum key distribution (QKD) algorithm. QKD is a network protocol that generates shared cryptographic keys between pairs of networked nodes by using both quantum signaling and classical messaging. The simulation carefully modeled the quantum signaling, but the classical networking was assumed to be ideal. It is well known, however, that tactical wireless networks are less than ideal. So, to evaluate the applicability of QKD to such networks, U.S. Army Combat Capabilities Development Command (DEVCOM) Army Research Laboratory (ARL) researchers wanted to see the impact of more realistic communication effects upon the QKD algorithm.

The situation of evaluating the expected behavior of some arbitrary software when put into an operational context is the core mission of the ARL Network Science Research Laboratory (NSRL). This task typically entails creating a cluster of virtual machines (VMs) that serve as proxies for Army platforms and then connecting them together in a virtual network whose performance is dictated by the Extendable Mobile Ad-Hoc Network Emulator (EMANE) system.[1] And so for the sake of compatibility with other ongoing projects, EMANE is the natural target for providing realistic communications to a QKD simulation.

EMANE is a network *emulator*, a distinction that contrasts with *simulation* primarily in focus. In network emulation a synthetic network interface is created on a host operating system (OS) that can be used like a normal, physical network interface. The focus then is on the "how" the network hosts behave for any given emulated network, rather than modeling how a network achieves its performance in the first place. In this context, the distinction is interesting because it led EMANE to adopt a distributed architecture wherein nodes exchange control and user data over a high-speed back-channel to perform the synthetic network's operations.

Alternatively, as typical with simulations, the QKD simulation is a standalone program written to model an algorithm's properties. Notably, this simulation uses a centralized architecture—all simulated entities are processed in a single program on a single computer. The major hurdle to EMANE/SQUANCH integration is in the resolution of these two very different approaches to network simulation: centralized and distributed.

To resolve the mismatch between the two architectures, two integration options present themselves: either distribute the QKD simulation communications or centralize the EMANE architecture. For the distributed approach, the simulation

would be run on one host and an EMANE network run on several others—with one EMANE host for each simulated node. Then, as depicted in Figure 1, whenever a node simulates classical messaging, the message would be forwarded to the corresponding EMANE node, sent over the EMANE network to its destination, and then back to the simulated receiver. The centralized approach is much the same, except in this case multiple EMANE instances could be run on the same host as the simulation.

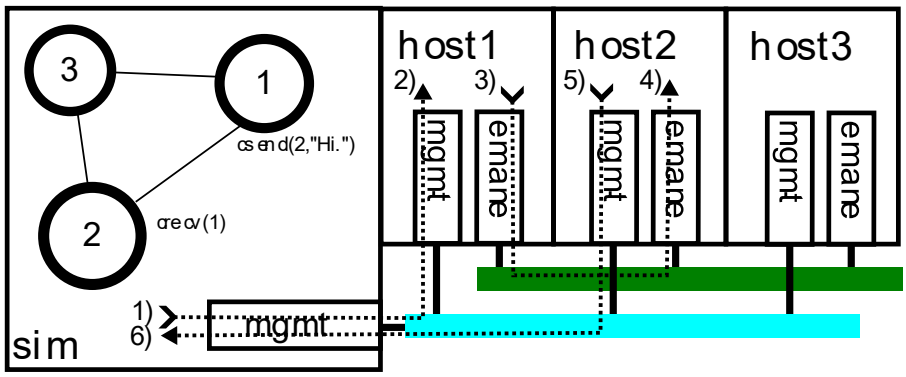

Figure 1. Depiction of forwarding a message over the EMANE network. 1) A message, "Hi." arrives at the management network interface (MGMT) from the simulation host (sim). 2) The message is sent to host 2 to arrive on its management interface. 3) Host 1 recognizes that the incoming message must be forwarded over the EMANE network, which provides the communication effects for the simulated 1 ↔ 3 link. On reception at 4), host 2 returns the message over the management interface 5) to the sim host 6), where it is passed to the waiting simulation.

Of the two options, the centralized approach, which we refer to as intra-host networking, seemed more direct and its design comprises the bulk of this report. The concept of intra-host routing is to initialize multiple EMANE network interfaces on an individual host and associate each interface with a corresponding simulated node. Operationally, a node would send messages out its interface, the message would be carried over the EMANE network until it appears at some other interface and then would be delivered back to the corresponding recipient in the simulation.

There is a problem with the intra-host concept. The EMANE software does allow multiple interfaces to be created on a single machine (allowing emulation of, for example, a single device belonging to multiple networks). However, when all interfaces belong to the same network, messages passed from the simulation will never traverse the EMANE network. This is because, as described in Figure 2, when a node, i, asks the OS to send a message to the IP address associated with some other node, j, the OS will recognize that it owns the destination address and will bypass the network and immediately pass the message to simulation node j listening at that address.

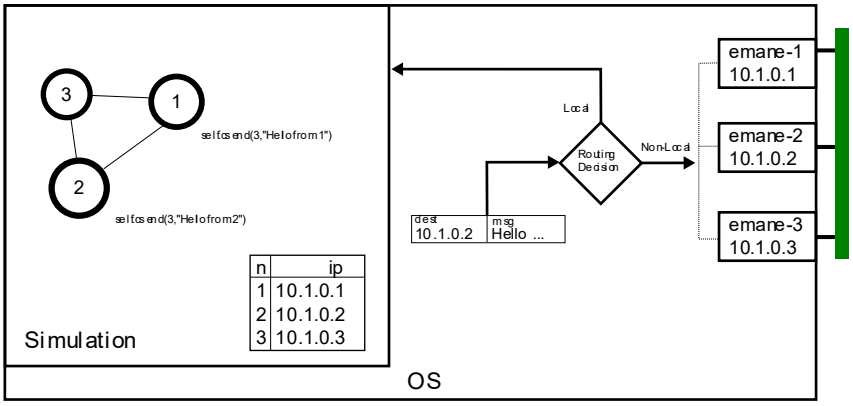

Figure 2. Depiction of a simulation attempting to send messages to EMANE interfaces. (Simulation) simulated nodes generate internet protocol messages. (OS) an outbound message for n = 2 OS will be returned to simulation through "Local" decision.

Another immediate problem with the intra-host concept is that, even if the OS could be persuaded to send a packet over the network, it does not have enough information to determine on which interface to send the message. A user program that wants to send a message only has a destination IP address to use; the routing setup of the host machine determines an outbound interface. And since multiple nodes should be allowed to message any other node, contradictory routing rules are implied.

Our innovation hinges on using the destination IP to effectively specify both a "to" and "from." This allows noncontradictory routing rules so that the kernel can select proper outbound interfaces. In concert with routing, low-level packet manipulations are made to change the source and destination addresses as messages enter or leave the network. As an example, Figure 3 shows the sequence of rewrites on a packet addressed to 10.1.1.3, which is being used in this case to represent the address of a message from node 1 (3rd octet) to node 3 (4th octet).

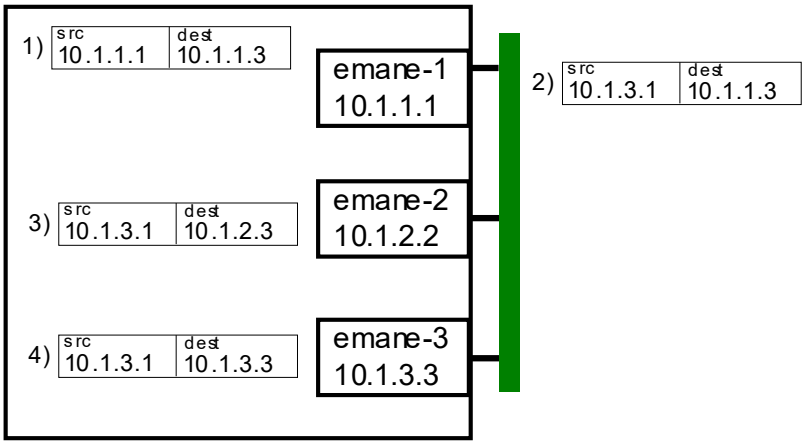

Figure 3. A sequence of source and destination address rewrites to route a packet. (1) The OS determines the outgoing interface for 10.1.1.0/24 messages is 10.1.1.1. In (2) the return address is modified as it is being put on the wire. Intermediate node, *emane-2*, facilitates routing by a change of destination address to *10.1.2.3* (2-to-3 address) and (3) the message is received on *emane-3* and the destination rewritten to match the interface (4).

Implementing the solution entails initiating routing rules that work with this addressing scheme. The rules, as well as a generalized Bash function, *link-ip*, that maps source-destination pairs to IP addresses are discussed in Section 2.3. Messages are put on the network with destinations that do not exist in the network (e.g., 10.1.1.3 vs. 10.1.3.3) and Section 2.4 shows how Address Resolution Protocol (ARP) rules can be used to ensure that messages are received at the appropriate interfaces. In Section 2.5, a network address translation (NAT) rule is introduced that performs the message destination rewriting and Section 2.6 gives the source rewriting rule and shows how source and destination rewrites can be used with additional routing rules to perform multiple hop routes. Finally, a few system-wide configuration options must be selected for the solution to work, as discussed in Section 2.7.

Intra-host IP networking is interesting in how its implementation illustrates fundamental networking technologies. In hindsight, however, it seems that distributing the simulation's communications may be the more sound approach. An outline for such a system is discussed in Section 3, and some benefits and infelicities of our adopted approach are discussed in the Conclusion (Section 4).

2. Intra-host IP Network

This section describes the design of intra-host routing in terms of the path a message takes as it traverses the network as in Figure 1. Operationally, a message generated in the simulation is passed to the host OS, to an EMANE network interface, to the EMANE link layer, to another EMANE interface, and finally back to the

simulation. At each step there is a decision or barrier that must be navigated, and the following sections describe those maneuvers.

2.1 Simulation Interface

The first step in—or perhaps a precursor to—intra-host routing, is getting messages out of the simulation. The QKD simulation is written in Python using the SQUANCH[2] framework. And although it is not designed to interface with the real world, an easy integration target did present itself.

SQUANCH provides classical communications primitives for sending and receiving classical messages, `csend()` and `crecv()`. The primitives are methods for the SQUANCH "`Agent`" class, so by providing a derived class "`AgentN`" that overrides only these methods, a simulation could be run in pure SQUANCH by initializing simulated nodes as members of the `Agent` class, or with EMANE enabled by using `AgentN` members.

Deriving via inheritance is a starting point though, and not the finished solution. The behavior of the implemented methods also needs to match the original in some regard. In the original implementation `csend()` would execute instantly, while `crecv()` would wait indefinitely until a message is received. Subsequently, the QKD protocol was written to expect—and ultimately depends upon—these behaviors or *semantics*.

The SQUANCH classical communication routines are natively modeled by passing messages through a memory queue. This underlying "queue" semantics determines the `csend/crecv` behavior in the ways we find important. So then, to ensure that the two implementations would share the same type of behavior, we implemented the `AgentA` methods based on the same underlying queue semantics. Specifically, by relying on the PUSH/PULL pattern of ZeroMQ[3,4] Python networking library, writing alternative implementations is straightforward.

Though it is true that our QKD simulation is written in Python and relies on ZeroMQ networking, those details are given for context, which are internal to the simulation. The ultimate interface to the intra-host routing system is the same as any other network—via IP addressing.

2.2 Device, Address, and Network Mappings

For the design of our solution, we begin by creating a few baseline functions that establish the relationship between simulated nodes and networking information. The implementation of these functions should be considered part of the network design process and not utilities that will work for arbitrary networks. Their basic

form is to return specific information when supplied with a simulation node identity. For simplicity, we identify the simulation nodes with the natural numbers $\mathbb{N}$ and summarize their applicability with function signatures in the following list.

- $dev : \mathbb{N} \to Name$
- $hw\text{-}ip : \mathbb{N} \to IP$
- $subnet : \mathbb{N} \to CIDR$
- $link\text{-}ip : \mathbb{N} \times \mathbb{N} \to IP$

In the above list, each simulation node is assigned a network interface and *dev* tells which node belongs to which EMANE interface. Similarly, each interface is assigned a single IP address by *hw-ip*, and subnetwork by *subnet*. The signature of *link-ip* specifies that it requires a set of two inputs, representing the "from" and "to" nodes.

As previously mentioned, the *link-ip* signature is the crux of our single host networking scheme. For a message going from simulation node $i{:}\mathbb{N}$ to $j{:}\mathbb{N}$ we generate a unique IP address, *link-ip i j*. There are N^2 *IP* addresses needed for any network simulation of size N and a cap of $N = 65{,}536$ for the whole IPv4 network space.

2.3 Default Routing

To achieve our goal of intra-host networking, we first ensure that when the OS receives a message addressed to *link-ip i j*, it selects the appropriate device, *dev i*, for transmission. This determination is made by the host's routing table, which is populated with addresses in classless inter-domain routing (CIDR) form (e.g., 10.1.1.0/24). For Linux, the routing rule can be set with the *ip* command in a form that relies on the baseline functions. Note in the following that *$(...)* is the Bash syntax for using the value of the resulting output of the command inside.

```
> ip route add $(subnet $i) dev $(dev $i)
```

The rule states that packets with addresses belonging to *subnet i* should be sent out *dev i*. For any destination *link-ip i j* to be selected for *dev i* it must be a member of *subnet i*. Membership is defined by sharing a common network prefix, and we can ensure consistency between *link-ip* and *subnet* by construction. For instance, our implementation defines *subnet* based on *link-ip* with *subnet i = link-ip i 0/CIDR_length*. This works because *link-ip i j* is constructed from a prefix based on *i* and suffix based on *j*, the suffix of *j = 0* is 0, and the bit-length of *i* is the *CIDR_length*.

The following implementation defines Bash functions for *subnet* and *link-ip* in terms of a network prefix, *cidr_prefix*, e.g., *10.1* (and *cidr_prefix_len*, e.g., 16). *link-ip* works by first converting the octet representation of the user-defined prefix, *cidr_prefix*, to a 32-bit integer with *prefix2int*. With *link2int*, an integer is derived by splitting the remaining bits and assigning the first half to the n-bit representation of *i* and the second half the n-bits of *j*. These two integers are then added together and converted back to the octal representation of that number with *int2ip*. *subnet* is then defined in terms of *link-ip _ 0* and *CIDR_length = 32 - (32 - cidr_prefix_len)/2*–an assignment that has not been mathematically reduced to preserve a 1-bit longer representation of *i* than *j* in the case of odd *cidr_prefix_len*. As a concrete example, defining *cidr_prefix_len=16* and *cidr_prefix=10.1* results in functions where *link-ip i j = 10.1.i.j* and *subnet i = 10.1.i.0/24*. And so, a message from 1 to 2 would be addressed "10.1.1.2," but a message from 3 to 2 would be "10.1.3.2."

```
link-ip(){
  declare from=$1
  declare to=$2
  declare n_subnet=$(( 32 - cidr_prefix_len ))
  declare prefix=$( prefix2int $cidr_prefix $cidr_prefix_len )
  declare suffix=$( link2int $n_subnet $from $to )
  int2ip $(( prefix + suffix ))
}

subnet(){
  echo $(link-ip $1 0)/$(( 32 - (32 - cidr_prefix_len)/2 ))
}
```

prefix2int takes an octal representation of a prefix, e.g., *10.1,* and associated length, e.g., 16, to a 32-bit integer. It first parses the input prefix to an array of integers (10.1 → (10 1)) with *ip2octets* (implementation omitted). The octets are converted to an integer with *octets2int* ((10 1) → 256 + 1 = 257) (omitted). The result is bit shifted left to occupy the leftmost prefix length bit (257/16 → 16842752).

```
prefix2int(){
  declare prefix=$1
  declare len=$2

  declare -a octs=( $(ip2octets $prefix) )
  declare n_octs len_diff
  n_octs="${#octs[@]}"
  len_diff=$(( (n_octs * 8) - len ))
  [[ $len_diff -lt 0 ]] && len_diff=0
```

```
  declare num=$(octets2int "${octs[@]}")
  echo $(( (num >> len_diff) << (8 * (4 - n_octs) + len_diff) ))

}
```

link2int takes three arguments, the number of bits in its address space, the "from" node, and the "to" node. It does some bounds error checking on the requested link against the bit limit but is otherwise just the simple addition of a leftward bit shifted "from" with an unshifted "to."

```
link2int(){
  declare subnet_n_bits=$1
  declare half_subnet=$(( $1 / 2 ))
  (( N  = 1 << half_subnet ))
  if [[ $2 -lt $N ]] && [[ $3 -lt $N ]]; then
    declare from=$(( $2 << half_subnet ))
    declare to=$3
    echo $(( from + to))
  else
    echo 0
    false
  fi
}
```

To ensure consistency with *link-ip* and the interface's hardware address *hw-ip*, we simply assign to it the address resulting from "sending a message to one's self," *hw-ip i = link-ip i i*:

```
hw_ip(){
  link_ip $1 $1
}
```

Finally, it is important to note that EMANE automatically initializes a default route based on the EMANE configuration. So instead of a call to *ip route*, we assign the transport definition appropriately, as shown in a notional section of a node's platform.xml, below. Therein, sections within curly braces "{}" are expected to be replaced by the values returned by the enclosed functions, or in the case of *cidr_mask,* the appropriate subnet mask associated with the device's subnet (e.g., *10.1.1.0/24* → *255.255.255.0*).

```
<nem definition="nem.xml" id="{i}">
  <transport definition="transport.xml">
    <param name="address" value="{hw-ip i}"/>
    <param name="device" value="{dev i}"/>
    <param name="mask" value="{cidr_mask}"/>
  </transport>
</nem>
```

2.4 Listening for Non-existent IPs

With the addressing and routing policy in place, properly addressed messages will now be passed to appropriate network interfaces for transport across the network. These interfaces are provided by the EMANE software and are not connections to actual WiFi or Local Area Networks (LANs). They do however mimic an Ethernet LAN to an extent–there is an Ethernet address associated with the interface–and so every interface can be viewed as being physically connected to an EMANE LAN.

At the Ethernet level of networking, messages are passed between devices by direct address. Hosts hear all messages on the wire but typically only process ones directly addressed to their own Ethernet address. And so, without further information, the interface will not know how to send outbound traffic.

At this level the ARP is used to glue the higher layer IP addresses to individual hosts on the network by keeping a table of Ethernet-to-IP resolutions. Normally, the protocol automatically determines these relationships by broadcasting requests that ask neighbors for these associations. In this case, however, no interface addresses, *hw-ip i = link-ip i i* will be associated with any destination *link-ip i j*.

The ARP table can be initialized manually with the *arp* command. The following function, *arp-link*, adds an ARP table entry for the one-way link between two nodes. Specifically, it says that outgoing messages on *dev i* addressed to *link-ip i j* can be sent to the hardware address of *dev j*. As a result, by populating the table with proper resolutions we can ensure that IP destinations are received at any particular interface.

```
arp-link(){
  declare dev_i=$( dev $1 )
  declare dev_j=$( dev $2 )
  declare link=$( link-ip $1 $2)
  sudo arp -i $dev_i -sD $link $dev_j
  }
```

2.5 Destination Translation

At this point in our description, we can route messages out interfaces for each simulated node, and ARP rules are used to ensure outbound messages arrive over the EMANE network at the proper destination interface. The messages will only be received if the EMANE network determines that the message is delivered based on the current, emulated conditions. Consequently, configuration of intra-host routing must be supported by the EMANE network, and for the sake of this discussion, we

assume that it is. So, given that the message does traverse the network, the receiving interface must then determine what to do with the incoming message.

Assuming the case that the message for *link-ip i j* has reached its destination at *dev j*, the kernel would see that it is not the owner of *link-ip i j* because all devices are bound to *link-ip j' j'*. However, if the destination address was instead *link-ip j j*, the kernel would see that the message has reached its destination and would pass it to a user space program listening at that address on *dev j*. NAT is a kernel technology that allows a router to rewrite message source and destination addresses. In particular, we use destination NAT (DNAT) rules that change the destination of messages as they enter the kernel from the network and *before* it makes a routing decision.

The function *nat-link-destination* calls the *iptables* function to set the conditions for performing a DNAT. It matches against the input device, *-i*, source address, *-s*, and destination, *-d*, in determining the address change. The application of *nat-link-destination i j [src] [dst]* specifies that a rule should be created for some link *i→j* that exists in the EMANE network. The rule checks that a message is received on *dev j* with a destination of *link-ip i j*, and with some source "*src*". If it matches, the destination is set to "*dst*". The optional *src* and *dst* parameters allow *nat-link-destination* to implement rules for both point-to-point connections and routed connections, where defaults are used in the case of direct links.

```
nat-link-destination(){
  declare src=${3:-$(link-ip $2 $1)}
  declare dst=${4:-$(hw-ip $2)}
  sudo iptables -t nat -A PREROUTING \
    -i $( dev $2 ) \
    -s $src \
    -d $( link-ip $1 $2) \
    -j DNAT --to-destination $dst
}
```

2.6 Return Address and Routing

At this point in the development of intra-host routing, we can successfully pass messages on direct connections from *i* to *j*. Messages can be passed through this network, and since we assume this means simulated nodes could pass messages on other's behalf, we have already implemented a theoretically complete network. However, there are two more essential features that make the network usable: return addressing and static routing.

Return addressing is not always necessary on a network. The User Datagram Protocol (UDP)—unreliable sibling of Transmission Control Protocol (TCP)—for instance, sends messages "connectionless." It does not require responses from the recipient. TCP, however, requires message receipt acknowledgments to be sent back to the source from the destination. With what has been presented so far, UDP messages can be passed, but TCP (and any protocol using TCP) cannot.

The problem is that EMANE-bound messages are given the wrong return addresses as they are put on the network. In Linux networking, each interface is bound to an IP address that is used as the return address for every message sent out on that interface. However, that return address is *link-ip i i*, while a message $i \rightarrow j$ needs the *link-ip j i*.

The solution we have implemented in *nat-link-source* uses Source NAT (SNAT) to change the return address as the packet is sent out onto the network. Specifically, if a message appears at *dev i* with outbound address *link-ip i j* we silently change its return address to *link-ip j i*. The receiving node can then properly address the return connection. Incidentally, this explains the default assignment of *src* in *nat-link-destination*.

```
nat-link-source(){
  sudo iptables -t nat -A POSTROUTING \
    -o $( dev $1 ) \
    -s $( hw-ip $1 ) \
    -d $(link-ip $1 $2) \
    -j SNAT --to-source $(link-ip $2 $1)
}
```

Finally, we want to allow interesting EMANE networks—networks that do not provide complete connectivity between every pair of nodes. In other words, networks that require multihop routing. Routing consists of a set of rules that tell a host which of their local neighbors should be used as relays in the next step to eventually reach a distant destination.

We implement routing through the introduction of both routing table rules and NAT table entries. Keeping on the topic of NAT, we implement the function, *nat-route*, that expects a path of nodes for which it will make NAT forwarding rules, e.g., *nat-route 1 2 .. N*. It makes a rule to set the return address on the original outgoing packet, a destination change rule for the middle nodes (*10.1.i-1.N* → *10.1.i.N*), and a destination transformation (as discussed) on the last node.

```
nat-route(){
  declare i mid_nodes last_node prev_node ret_addr
  last_node="${!#}"
  mid_nodes=( "${@:2: $(( $# - 2 ))}" )
```

```
  ret_addr=$( link-ip $last_node $1 )

  # Set return address of outgoing packet
  nat-link-source $1 $last_node

  # Intermediates change destination, 10.1.1.3 -> 10.1.2.3
  declare prev_node="$1"
  for i in ${mid_nodes[@]}; do
    next_hop="$( link-ip $i $last_node )"
    nat-link-destination $prev_node $last_node $ret_addr $next_ho
p
    prev_node="$i"
  done
  # set final destination to be this hw address
  nat-link-destination $prev_node $last_node $ret_addr
}
```

The NAT rules above support routing packets, but actual IP routing rules are also needed. The routing table rules encapsulated by *route-forwarding* tell the kernel when it can deliver a message bound for node k by first routing to j from i, where i and j are directly connected.

```
route-forwarding(){
  declare next_node current_dest gw current_dev path_tail current
_node

  last_node="''${!#}"
  current_node="$1"; shift

  for next_node in "$@"; do
    current_dest="$( link-ip $current_node $last_node )"
    gw="$(  link-ip $current_node $next_node )"
    current_dev="$( dev $current_node)"

    if [[ $current_dest != $gw ]]; then
      sudo ip route add $current_dest via $gw dev $current_dev 2>
/dev/null
    else
      break
    fi
    current_node="$next_node"
  done
}
```

2.7 Enable Kernel Routing

Some additional basic prerequisites are needed to enable the functionality discussed in this report. There are three kernel configuration options that must be set

appropriately. We provide the function *enable-forwarding*, which expects as arguments the list of EMANE interfaces that comprise the network.

The function works as follows. First, we enable routing, which allows the forwarding of packets received on one interface but bound for another. Second, we enable proxy ARP, which lets the interfaces share the *link-ip* addresses that they will respond to, even though those devices are not actually on the network. Finally, reverse packet filtering is disabled for each of the supplied interfaces. This tells the kernel not to drop packets just because the destination is unreachable.

```
enable-forwarding(){
  # Enable forwarding in kernel
  echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward > /dev/null
  # Enable Proxy Arp
  echo 1 | sudo tee /proc/sys/net/ipv4/conf/all/proxy_arp > /dev/
null
  # Disable reverse packet filtering on configured interfaces.
  for iface in all $@ ; do
    echo 0 | sudo tee /proc/sys/net/ipv4/conf/$iface/rp_filter >
/dev/null
  done
}
```

Note that while there are ways to enable the settings on a more permanent fashion, the changes made by *enable-forwarding* are lost with every time the EMANE interfaces come down or the computer is rebooted. As such, *enable-forwarding* should be used as part of the network setup.

2.8 Network Initialization

We start the intra-host network by starting each network interface and then initializing the routing and NAT rules along specific paths. *start-network* depends on two variables, the size of the network, *N*, and a listing of paths in the network, *PATHS*. *PATHS* comprises multiple lines, with each line specifying an individual route.

The declaration of *PATHS* is another network design activity. Care must be taken, for instance, that any routing decision should be unambiguous, and so two different paths with the same endpoints, for example, *PATHS="1 2 3\n1 3"*, should not be allowed. Also, we should generally ensure both "to" and "from" paths are established for any pair of connected nodes. Finally, a path "*1 2 3*" implies both subpaths "*1 2*" and "*2 3*". However, initializing the longer path does not enable direct communications on the shorter using this scheme. So, when initializing a multihop path, all subpaths should also be included.

```
start-network(){
  for n in $(seq $N); do
    platform=$(dev $n)
    emane start $platform.xml
  done
  echo "$PATHS" | while IFS= read path; do
    route-forwarding $path
    nat-route $path
  done
}
```

Our general *PATHS* design solution is to first take the network graph of the EMANE communications network and find some spanning tree. Then, for each end-point node (only one neighbor in tree), include the paths for each of the other end-points. Then, for each of these paths include all subpaths. Then for all the paths, include the reverse path. A simple example is given in Figure 4.

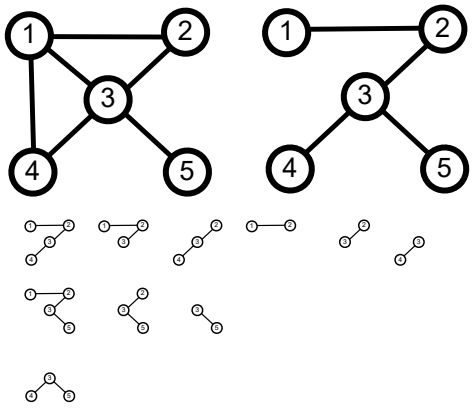

Figure 4. A simple five-node network (top left), the chosen spanning tree (top right), and the individual routes (bottom) that must be initialized to achieve full routing connectivity.

3. Alternative Approach: Distribute Simulation Messaging

The design of EMANE facilitates large-scale testing of network applications. This typically entails creating a cluster of VMs (or containers) wherein each VM has one or more EMANE interfaces. The VMs are connected via a control network that EMANE uses to coordinate the simulation of the emulated network and to tunnel EMANE-bound messages through.

The benefit of both virtualization technologies is that they provide isolation; VMs have different interfaces, routing tables, and others, from each other and their hosts. Accordingly, the extensive NAT, ARP, and routing considerations discussed in this report are not required. Therefore it is that same isolation that adds a barrier to simulated nodes' use of the EMANE network. Simulation nodes on one host cannot

directly use the network interface of another host. Instead, they must ask the foreign host to forward a message on their behalf.

The architecture to facilitate distributed simulation messaging is shown in Figure 1. As mentioned in the introduction, the simulation is run in one host of a virtual cluster that also includes *N* other hosts–each of which is mapped to one of the *N* simulated nodes. All *N+1* hosts are connected by a management network and the *N* virtual hosts are also interconnected via the EMANE network.

As with the centralized case, the simulation's *csend*/*crecv* would use a network programming library to enable sending simulation messages out through the networking stack. Simulated node *i*'s message to *j* would be sent to VM *i*, which would then forward it over the EMANE network to VM *j*. VM *j* would in turn return the message back to the simulation VM and simulated node *j*.

In this architecture, the network of EMANE VMs is reachable over some management interface, *mgmt*. Ostensibly, each simulation node would always address its corresponding EMANE VM on *mgmt*, but to forward the message the destination node must be encoded somehow. In keeping with the intra-host solution, however, the source and destination might be encoded through *link-ip i j*-based addressing. Then, routing rules on the host would forward messages to the appropriate VM and appropriate rules would be implemented on each VM to forward messages from the *mgmt* network to the *emane* network and back again. The routing situation is improved, since translation only exists at the interface and not between every hop on every path.

4. Conclusion

This report introduces a novel approach to intra-host routing, which is a method of configuring a Linux-based OS such that IP packets can be bounced between network interfaces to simulate a network. We facilitate this utility through manipulations to the kernel routing table, manual entries in the ARP lookup table, and NAT rules. The operation of the network depends on adopting a convention of encoding source and destination into each IP address, as modeled by *link-ip*, and for which we supply a general-purpose implementation. As an alternative to intra-host routing, we outline how similar functionality can be implemented with virtualization technology—VMs or containers.

Of the two implementation options, intra-host routing was initially attractive because it lacks the extra messaging incurred by the containerized option. It also does not require the extra management overhead of initializing and deploying a virtual cluster. Additionally, the novel addressing allows easy access to the intra-

host network for any networking technology. For instance, a ping can be sent across any path just by specifying the appropriate address, e.g., *ping $(link-ip i j)*.

There are multiple issues to address with our approach as well. For one, the amount of routing and NAT rules can quickly grow large. In the emulation of a 10-node metropolitan area network, on the order of 150 routing rules and over 1000 NAT rules were needed. Given that these rules need to be referenced on a per flow basis, it is possible that this rate of growth may quickly outweigh any latency delays that would otherwise be incurred by the distributed approach. Another problem is that this nonstandard approach prohibits using ad hoc routing solutions, such as OLSR. In fact, this solution has only been shown to work for static networks with predetermined topology.

Moving to the container-based deployment should alleviate many of the burdens of the intra-host implementation. When a message is forwarded from the simulation host to the EMANE VM, it is translated from a *link-ip* on the *mgmt* network to a standard IP on the *emane* network. As such, normal routing solutions are applicable, and dynamic networks and routing become a simulation option.

5. References

1. [EMANE] Extendable Mobile Ad-hoc Network Emulator [homepage]. Adjacent Link LLC; c2025. https://adjacentlink.com/documentation/emane/v1.0.1/

2. Bartlett B. A distributed simulation framework for quantum networks and channels. arXiv. 2018. arXiv:1808.07047. https://doi.org/10.48550/arXiv.1808.07047

3. Granger BE, Ragan-Kelley M. PyZMQ: Python bindings for ØMQ. Pythonfix; 2022 Jan 22. https://pythonfix.com/pkg/p/pyzmq/

4. ZeroMQ: an open-source universal messaging library [homepage]. The ZeroMQ authors; c2025. https://zeromq.org/

List of Symbols, Abbreviations, and Acronyms

ARL	Army Research Laboratory
ARP	Address Resolution Protocol
CIDR	classless inter-domain routing
DEVCOM	U.S. Army Combat Capabilities Development Command
DNAT	destination NAT
EMANE	Extendable Mobile Ad-hoc Network Emulator
IP	Internet Protocol
LAN	Local Area Network
MGMT	management network interface
NAT	network address translation
NSRL	Network Science Research Laboratory
OLSR	Optimized Link-State Routing
OS	operating system
QKD	quantum key distribution
sim	simulation
SNAT	Source NAT
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	virtual machine

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLB CI TECH LIB