Local Cell-Based Emulation Framework

Sahas Munamala

The goal of this system is to provide a lightweight yet realistic environment for testing and developing reliable networking protocols (e.g., Alternating Bit, tree-structured transaction routing) with the same abstractions used in production distributed systems. While initially designed for single-machine emulation, the architecture is intentionally modular and forward-compatible with cross-machine and hardware-accelerated deployment models.

This specification covers:

- The lifecycle and responsibilities of a **Cell**
- The topology control and orchestration handled by the **Datacenter**
- The data model for ports, links, and runtime reconfiguration
- Best practices for scaling, observability, and extensibility

Background

For now, a single instance of a cell is generated in the shell where the python3 command is run.

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Alternating Bit Protocol Implementation')
    parser.add_argument('--config_file', '-c', type=str, help='Path to the YAML configuration file')
    args = parser.parse_args()

    if args.config_file:
        config = load_config(args.config_file)
        sim = Sim.from_config(config)
        asyncio.run(sim.run())
```

The sim config contains the following information:

```
config:
  protocol: tree             # does nothing currently, but lets you choose base protocol
  log_dir: /opt/hermes/logs  # log directory
  log_level:
    websocket_server: info
    alice: info
    charlie: info
  node_id: sahas
ports:
  - name: en2
```

```
      interface: en2
    - name: en3
      interface: en3
    - name: en4
      interface: en4
```

Design Principles

- **Single responsibility**: each `Cell` is an isolated, controllable unit with no implicit global state
- **Explicit orchestration**: all topology is created and torn down via the `DatacenterController`
- **Dynamic reconfigurability**: cells and links can be added/removed at runtime via XML-RPC
- **Scalability-conscious**: uses UNIX domain sockets to avoid OS port exhaustion and maximize emulation scale
- **Replaceable transport abstraction**: link setup is abstracted to allow future replacement (e.g., real UDP or TCP)

Cells

A cell is represented by one process on the machine. It encompasses all port threads, and has a unique datacenter identifier.

- **A cell is defined at the datacenter level**
- **A cell is indistinguishable from another cell** other than by its datacenter unique id
- **A cell has a fixed logging configuration**

Each cell needs the following information to run

1. The ports of the cell, and what ports/interfaces to bind on
2. Cell logging configuration

If cell configuration needs to change, remove the cell, then create another one in it's place.

A cell will be controlled by the datacenter with a python xmlrpc.server

```python
class Cell:
    def __init__(self):
        self.bound_ports = {}  # e.g., {"en2": socket_obj}
        self.config = {}       # config per port
        self.running = True

    def bind_port(self, port_name, port_config):
        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        sock.bind((udp_port_or_interface, 0))  # bind to any free port
        self.bound_ports[port_name] = sock
        return f"Bound {port_name} to {udp_port_or_interface}"

    def unbind_port(self, port_name):
        if port_name in self.bound_ports:
            self.bound_ports[port_name].close()
            del self.bound_ports[port_name]
            return f"Unbound {port_name}"
        return f"{port_name} not found"

    def link_status(self, port_name):
        if port_name not in self.bound_ports:
            return "unbound"

        # You can make this richer later with packet counters, timestamps,
etc.
        return "connected"

    def shutdown(self):
        for port in self.bound_ports.values():
            port.close()
        self.running = False
        threading.Thread(target=self._delayed_exit).start()
        return "Shutting down..."

    def _delayed_exit(self):
        time.sleep(0.5)
        sys.exit(0)

    def heartbeat(self):
        return "alive"
```

Links

A link does not have an active physical analog in the emulation. Instead, it's a logical connection of state machines through a udp connection. I'm thinking the cell process must be instructed which interface to bind on. This is the only way to build in dynamic reconfiguration of the topology.

- A link is defined at the datacenter level as a tuple of (cell1_id:port, cell2_id:port)
- A link ber should be defined (0->1 float)
- Can add/remove links during emulation

If link configuration needs to change, remove the link, then re-bind the ports.

UNIX Domain Sockets

Previously, when creating local process-to-process communication I was using UDP sockets. These are cumbersome and have a fixed limit on the number of ports you can use on the system. While we could continue to use them, I would much rather prefer to use UNIX DGRAM sockets under the hood. This makes the emulation more scaleable, without having to change the protocol abstractions at all.

To connect, you would have to create the socket individually based on the unix socket path. See docs for more details.

Something like

```
async def setup_unix_socket(port_config, protocol_factory):
    """
    Creates a UNIX domain socket transport/protocol pair.

    Parameters:
        port_config: dict with 'unix_socket' key
        protocol_factory: lambda returning asyncio.Protocol
    Returns:
        (transport, protocol)
    """
    loop = asyncio.get_running_loop()
    sock_path = port_config["unix_socket"]["path"]
    mode = port_config["unix_socket"]["mode"]

    # Clean up stale socket
    if mode == "server" and os.path.exists(sock_path):
        os.unlink(sock_path)

    if mode == "server":
        server = await loop.create_unix_server(
            protocol_factory, path=sock_path)
        print(f"[UNIX] Listening as server on {sock_path}")
        return server  # return the server object for shutdown later

    elif mode == "client":
        transport, protocol = await loop.create_unix_connection(
            protocol_factory, path=sock_path)
        print(f"[UNIX] Connected as client to {sock_path}")
        return transport, protocol

    else:
        raise ValueError("Invalid mode: must be 'server' or 'client'")
```

Interfaces

When binding to an interface, we will use the existing link-bringup code to isolate one side as the client and the other side as the server.

```
while True:
        # For real ethernet interfaces, use IPv6 neighbor discovery
        result = await get_ipv6_neighbors(self.config.interface)
        if result:
            self.is_client, self.remote_addr, self.local_addr = 
self.extract_running_details(result)
            self.logger.info(f"Found Neighbor. result={result}")
            #...
            transport, self.protocol_instance = await 
self._loop.create_datagram_endpoint(
                lambda: self.protocolClass(
                    io=self.io,
                    name=self.config.port_id,
                    sending_addr=remote_addr,
                    is_client=is_client,
                    faultInjector=self.faultInjector
                ),
                remote_addr=remote_addr if is_client else None,
                local_addr=local_addr,
                reuse_port=True
            )
    await asyncio.sleep(1)  # Add delay between retries
```

Port Configuration Data Model

```
# port_config is a dict mapping string <-> string, because xmlserver
forces you to send raw python types for serialization simplicity.
port_config = {
    "port_name": "p0",              # Logical name of the port inside the
cell

    # Option 1: Bind to a real or virtual interface (e.g., en0, tun0,
dummy)
    "interface": "en2",              # Optional: Bind to this interface
using UDP

    # Option 2: Use a UNIX domain socket
    "unix_socket": {
        "path": "/tmp/link_sahas_alice.sock",  # Filesystem path to the
socket
        "mode": "server"                        # "server" or "client"
    },

    # Optional parameters for either mode
    "ber": 0.01,                     # Bit error rate (unimplemented)
    "log_level": "error"             # Per-port logging
}
```

Datacenter

The 'datacenter' is local to one computer. It is an active central controller of all the cells and links emulated on one machine.

- Needs unique identifiers/distinguishability for each cell:port
- Can add/remove cells and links during emulation

```
class DatacenterController:
    def __init__(self):
        self.cells = {}        # cell_id -> xmlrpc proxy
        self.links = {}        # link_id -> ((cell1, port1), (cell2,
port2))

    def add_cell(self, cell_id, rpc_url):
        self.cells[cell_id] = xmlrpc.client.ServerProxy(rpc_url)
        print(f"[DC] Added cell {cell_id}")

    def remove_cell(self, cell_id):
        if cell_id not in self.cells:
            print(f"[DC] Cell {cell_id} not found")
            return

        # Remove links involving this cell
        to_remove = [lid for lid, ((c1, _), (c2, _)) in self.links.items()
                     if c1 == cell_id or c2 == cell_id]
        for lid in to_remove:
            self.remove_link(lid)

        try:
            self.cells[cell_id].shutdown()
        except Exception as e:
            print(f"[DC] Warning: failed to shutdown {cell_id}: {e}")
        del self.cells[cell_id]
        print(f"[DC] Removed cell {cell_id}")

    def add_link(self, link_id, cell1_id, port1, cell2_id, port2,
sock_path):
        server_cfg = {
            "port_name": port1,
            "unix_socket": {"path": sock_path, "mode": "server"}
        }
        client_cfg = {
            "port_name": port2,
            "unix_socket": {"path": sock_path, "mode": "client"}
        }

        self.cells[cell1_id].bind_port(port1, server_cfg)
        self.cells[cell2_id].bind_port(port2, client_cfg)

        self.links[link_id] = ((cell1_id, port1), (cell2_id, port2))
        print(f"[DC] Added link {link_id}: {cell1_id}:{port1} ↔
{cell2_id}:{port2}")

    def remove_link(self, link_id):
        if link_id not in self.links:
            print(f"[DC] Link {link_id} not found")
            return

        (c1, p1), (c2, p2) = self.links[link_id]
        self.cells[c1].unbind_port(p1)
        self.cells[c2].unbind_port(p2)
        del self.links[link_id]
        print(f"[DC] Removed link {link_id}")
    def get_cell_status(self, cell_id):
        try:
            return self.cells[cell_id].heartbeat()
        except Exception:
            return "unreachable"

    def get_link_status(self, link_id):
        if link_id not in self.links:
            return "unknown"

        (c1, p1), (c2, p2) = self.links[link_id]
        try:
            s1 = self.cells[c1].link_status(p1)  # new RPC hook inside
cell
        except Exception:
            s1 = "unreachable"

        try:
            s2 = self.cells[c2].link_status(p2)
        except Exception:
            s2 = "unreachable"

        return {"endpoints": [(c1, p1), (c2, p2)], "status": [s1, s2]}

    def get_topology(self):
        return {
            "cells": list(self.cells.keys()),
            "links": {
                lid: {"endpoints": [f"{c1}:{p1}", f"{c2}:{p2}"]}
                for lid, ((c1, p1), (c2, p2)) in self.links.items()
            }
        }
```

Example usage:

```
dc = DatacenterController()

# Add two cells
dc.add_cell("sahas", "http://localhost:9001")
dc.add_cell("alice", "http://localhost:9002")

# Add a link
dc.add_link(
    "link_1",
    cell1_id="sahas",
    port1_name="en0",
    cell2_id="alice",
    port2_name="en1",
    sock_path="/tmp/sahas_alice.sock"
)

# Query status
```

```
print(dc.get_cell_status("sahas"))
print(dc.get_link_status("link_1"))
print(dc.get_topology())

# Tear down
dc.remove_link("link_1")
dc.remove_cell("sahas")
dc.remove_cell("alice")
```

Physical Node Emulation Model

This framework is designed to support both fully simulated multi-cell topologies and real-world testing on physical machines. To bridge the two, we treat each **physical node** as a **local datacenter with a single cell**, connected directly to its physical or virtual network interfaces.

🧩 Extensibility Opportunities

- Add packet-level metrics to `link_status()` (e.g., bytes sent, timestamp of last receive)
- Support interface-based links using IPv6 ND for auto client/server role detection
- Inject latency and BER with a `FaultInjector` wrapper per port
- Add CLI or REST tooling to wrap `DatacenterController`
- Enable batched topology updates (e.g., `apply_topology(version=X)` or `update_links(...)`) to reduce race conditions.

⏭️ Future Work (Not in Scope)

- Cross-machine coordination or remote datacenter federation
- Secure authenticated RPC between components
- Live migration of ports or cells between nodes
- Automated failure recovery or retry policies on bind/connect failure

⚠️ Cells Are Not Observable Internally (Yet)

Observation:
Cells expose `heartbeat()` and `link_status()`, but they don't provide visibility into internal protocol behavior, packet flow, or counters.

Why it's a risk:

- Without visibility into state machines, retransmission windows, or in-flight packets, debugging protocol failures becomes difficult.
- Test harnesses and integration pipelines cannot make assertions beyond binary connectivity.
- Performance metrics (e.g., throughput, delay, loss) cannot be collected or visualized.

Suggested Evolution:

- Add a `get_metrics()` or `get_port_stats()` method to each cell that returns counters like packets sent/received, bytes transferred, last activity timestamp, etc.
- Introduce a debug `event_log` buffer per port to record protocol-level events (e.g., TX/RX, timeouts, state transitions).
- Add tracing hooks to simulate packet path tracking, allowing high-level tools or dashboards to visualize flow activity.
- Consider exposing these metrics over a Prometheus-compatible or JSON-over-HTTP endpoint in the future.