```
#!/bin/bash
# Title: Chimera NOS N-Dimensional Tensor Simulation
#
# This script creates a standalone environment for simulating N-Dimensional
# "Hypercube" data exchange between Chimera Nodes.
#
# Features:
# - 4D Tensor Generation (Simulating Spatiotemporal Neural Weights)
# - Dynamic Shape Serialization/Deserialization
# - 64-bit Float Precision
#
# Execution:
# 1. chmod +x chimera_ndim_sim.sh
# 2. ./chimera_ndim_sim.sh
#
################################################################################

ARCHIVE_DIR="chimera_ndim_sim"

echo "============================================================"
echo "Chimera NOS N-Dimensional Architecture Setup"
echo "------------------------------------------------------------"
echo "Extracting environment..."
echo "============================================================"

mkdir -p "$ARCHIVE_DIR"

# 1. Extract Protocol Library
cat << 'EOF_PROTO' > "$ARCHIVE_DIR/adi_protocol.py"
import struct
from typing import Any, Dict

PROTOCOL_HEADER_SIZE = 13
FIXED_U32_SIZE = 4

class ADIProtocolError(Exception):
    pass

def write_u32_be(value: int, buffer: bytearray, offset: int) -> None:
    data = struct.pack('!I', value)
    buffer[offset:offset + FIXED_U32_SIZE] = data

def read_u32_be(buffer: bytes, offset: int) -> int:
    if len(buffer) < offset + FIXED_U32_SIZE:
```

```
        raise ADIProtocolError("Buffer too short to read u32_be.")
    return struct.unpack('!I', buffer[offset:offset + FIXED_U32_SIZE])[0]

def create_payload_header(payload_size_bytes: int) -> bytes:
    return struct.pack('!I', payload_size_bytes)
EOF_PROTO

# 2. Extract Server (Tensor Aware)
cat << 'EOF_SERVER' > "$ARCHIVE_DIR/Server_GraphicsBBS.py"
import socket
import struct
import threading
import logging
import adi_protocol as adi

HOST = '0.0.0.0'
PORT = 12345

DATA_TYPE_NEURAL_TENSOR = 8 # NEW: N-Dimensional Tensor Type
ORIGIN_DOMESTIC = 1
ACCESSIBILITY_NONE = 0

MSG_TYPE_ID_ASSIGNMENT = 255
MSG_TYPE_LARGE_FILE_BROADCAST = 103
REQ_TYPE_NX_PUSH = 21

logging.basicConfig(level=logging.INFO, format='%(asctime)s - [CHIMERA C2] - %(message)s')

class Server:
    def __init__(self, host, port):
        self.host = host
        self.port = port
        self.clients = []
        self.client_lock = threading.Lock()
        self.next_id = 0

    def broadcast(self, message, sender_socket=None):
        with self.client_lock:
            for c in self.clients:
                if c['socket'] != sender_socket:
                    try:
                        c['socket'].sendall(message)
                    except: pass
```

```
def handle_client(self, conn, addr):
    cid = -1
    try:
        with self.client_lock:
            cid = self.next_id
            self.next_id += 1
            self.clients.append({'socket': conn, 'id': cid})

        logging.info(f"Node {cid} connected from {addr}")
        conn.sendall(struct.pack('!BI', MSG_TYPE_ID_ASSIGNMENT, cid))

        while True:
            header = self._recv_all(conn, 1)
            if not header: break
            opcode = struct.unpack('!B', header)[0]

            # Read & discard standard payload size (handled internally by logic)
            self._recv_all(conn, 4)

            if opcode == REQ_TYPE_NX_PUSH:
                self._handle_nx_push(conn, cid)
    except Exception as e:
        logging.error(f"Error with Node {cid}: {e}")
    finally:
        conn.close()

def _handle_nx_push(self, conn, cid):
    # Read Metadata: FileSize(Q=8), HashLen(I=4)
    meta = self._recv_all(conn, 12)
    total_size, hash_len = struct.unpack('!QI', meta)
    nc_hash = self._recv_all(conn, hash_len).decode('utf-8')

    # Read Name
    name_len = struct.unpack('!I', self._recv_all(conn, 4))[0]
    nc_name = self._recv_all(conn, name_len).decode('utf-8')

    # --- N-DIMENSIONAL METADATA PARSING ---
    # Read Rank (Number of Dimensions)
    rank = struct.unpack('!I', self._recv_all(conn, 4))[0]

    # Read Shape Array (Rank * 4 bytes)
    shape_bytes = self._recv_all(conn, rank * 4)
    shape = struct.unpack(f'!{rank}I', shape_bytes)
```

```
        logging.info(f"Incoming N-Dim Tensor from Node {cid}:")
        logging.info(f"  Name: {nc_name}")
        logging.info(f"  Rank: {rank}-Dimensional")
        logging.info(f"  Shape: {shape}")
        logging.info(f"  Size: {total_size} bytes")

        # Receive Tensor Data
        data = self._recv_all(conn, total_size)

        # --- BROADCAST ---
        # Reconstruct Metadata for Broadcast
        # Header: NameLen(I) + Size(Q) + Type(B) + Org(B) + Acc(B)
        base_meta = struct.pack('!IQBBB', len(nc_name), total_size,
DATA_TYPE_NEURAL_TENSOR, ORIGIN_DOMESTIC, ACCESSIBILITY_NONE)

        # Tensor Header: Rank(I) + Shape(I...)
        tensor_header = struct.pack('!I', rank) + shape_bytes

        payload = struct.pack('!I', cid) + base_meta + tensor_header + nc_name.encode('utf-8') +
data

        header = bytearray(13)
        adi.write_u32_be(cid, header, 8)
        header[12] = MSG_TYPE_LARGE_FILE_BROADCAST

        full_msg = header + adi.create_payload_header(len(payload)) + payload
        self.broadcast(full_msg, conn)
        logging.info(f"Broadcasted Hypercube Tensor to mesh.")

    def _recv_all(self, sock, n):
        data = b''
        while len(data) < n:
            packet = sock.recv(n - len(data))
            if not packet: return None
            data += packet
        return data

    def start(self):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind((self.host, self.port))
        s.listen(5)
        logging.info(f"Chimera N-Dim Server listening on {self.port}")
        while True:
```

```
            conn, addr = s.accept()
            threading.Thread(target=self.handle_client, args=(conn, addr), daemon=True).start()

if __name__ == "__main__":
    Server(HOST, PORT).start()
EOF_SERVER

# 3. Extract N-Dimensional Simulator Client
cat << 'EOF_CLIENT' > "$ARCHIVE_DIR/Client_Chimera_Sim.py"
import socket
import struct
import time
import sys
import threading
import numpy as np
import adi_protocol as adi

HOST = '127.0.0.1'
PORT = 12345

REQ_TYPE_NX_PUSH = 21
MSG_TYPE_LARGE_FILE_BROADCAST = 103
MSG_TYPE_ID_ASSIGNMENT = 255

def recv_all(sock, n):
    data = b''
    while len(data) < n:
        packet = sock.recv(n - len(data))
        if not packet: raise EOFError
        data += packet
    return data

def run_receiver(sock, my_name):
    print(f"[{my_name}] Receiver thread active.")
    while True:
        try:
            header = recv_all(sock, 13)
            length = struct.unpack('!I', recv_all(sock, 4))[0]
            payload = recv_all(sock, length)

            packet_type = header[12]

            if packet_type == MSG_TYPE_LARGE_FILE_BROADCAST:
                offset = 0
```

```
            sender_id = struct.unpack('!I', payload[0:4])[0]; offset+=4
            name_len, size, dtype, org, acc = struct.unpack('!IQBBB', payload[offset:offset+15]);
offset+=15

            # Read Tensor Header
            rank = struct.unpack('!I', payload[offset:offset+4])[0]; offset+=4
            shape = struct.unpack(f'!{rank}I', payload[offset:offset+(rank*4)]); offset+=(rank*4)

            filename = payload[offset:offset+name_len].decode('utf-8'); offset+=name_len
            raw_data = payload[offset:offset+size]

            # Reconstruct N-Dimensional Array
            arr = np.frombuffer(raw_data, dtype=np.float64)
            reshaped_arr = arr.reshape(shape)

            print(f"\n[{my_name}] >>> INCOMING TENSOR <<<")
            print(f"[{my_name}] Source: Node {sender_id}")
            print(f"[{my_name}] Name: {filename}")
            print(f"[{my_name}] Topology: {rank}-Dimensional Hypercube")
            print(f"[{my_name}] Shape: {shape}")
            print(f"[{my_name}] Data Verification:")
            print(f"    Sum: {np.sum(reshaped_arr):.2f}")
            print(f"    Mean: {np.mean(reshaped_arr):.2f}")
            print(f"    Corner Value [0,0,0,0]: {reshaped_arr[0,0,0,0]:.2f}")
            print(f"[{my_name}] >>> END TRANSMISSION <<<\n")

        except Exception as e:
            print(f"[{my_name}] Error: {e}")
            break

def client_main(mode):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        sock.connect((HOST, PORT))
        id_packet = recv_all(sock, 5)
        msg_type, my_id = struct.unpack('!BI', id_packet)
        name = f"Node_{my_id}_{mode}"
        print(f"[{name}] Online.")

        threading.Thread(target=run_receiver, args=(sock, name), daemon=True).start()

        if mode == "SENDER":
            time.sleep(2)
            print(f"[{name}] Generating 4D Hypercube Tensor...")
```

```
            # Generate 4D Tensor (Time, Depth, Height, Width)
            # Shape: (5 TimeSteps, 10 Depth, 10 Height, 10 Width) = 5000 elements
            shape = (5, 10, 10, 10)
            rank = len(shape)

            # Fill with identifiable data
            tensor = np.random.rand(*shape).astype(np.float64)

            data_bytes = tensor.tobytes()
            total_size = len(data_bytes)

            nc_name = "Hypercube_Weights_v4.tensor"
            nc_hash = "HASH_NDIM_TEST"

            # Build Payload
            # 1. Standard Metadata
            meta = struct.pack('!QI', total_size, len(nc_hash)) + nc_hash.encode('utf-8')
            meta += struct.pack('!I', len(nc_name)) + nc_name.encode('utf-8')

            # 2. Tensor Metadata (Rank + Shape)
            tensor_meta = struct.pack('!I', rank) + struct.pack(f'!{rank}I', *shape)

            # 3. Combine
            payload = meta + tensor_meta + data_bytes

            msg = struct.pack('!B', REQ_TYPE_NX_PUSH) +
adi.create_payload_header(len(payload)) + payload
            sock.sendall(msg)
            print(f"[{name}] Transmitted {rank}D Tensor (Shape {shape}).")

            time.sleep(5)

        elif mode == "RECEIVER":
            print(f"[{name}] Awaiting N-Dimensional data...")
            time.sleep(10)

    except Exception as e:
        print(f"Client error: {e}")
    finally:
        sock.close()

if __name__ == "__main__":
    mode = sys.argv[1] if len(sys.argv) > 1 else "RECEIVER"
```

```
    client_main(mode)
EOF_CLIENT

# 4. Create Orchestrator
cat << 'EOF_RUN' > "$ARCHIVE_DIR/run_simulation.py"
import subprocess
import time
import sys

print("--- CHIMERA N-DIMENSIONAL SIMULATION ---")

print("[ORCH] Starting C2 Server...")
server_proc = subprocess.Popen([sys.executable, "Server_GraphicsBBS.py"])
time.sleep(2)

print("[ORCH] Starting Receiver Node (B)...")
recv_proc = subprocess.Popen([sys.executable, "Client_Chimera_Sim.py", "RECEIVER"])
time.sleep(1)

print("[ORCH] Starting Sender Node (A)...")
send_proc = subprocess.Popen([sys.executable, "Client_Chimera_Sim.py", "SENDER"])

try:
    send_proc.wait()
    recv_proc.wait()
except KeyboardInterrupt:
    pass

print("[ORCH] Stopping Mesh...")
server_proc.terminate()
print("--- SIMULATION COMPLETE ---")
EOF_RUN

# 5. Final Instructions
echo ""
echo "------------------------------------------------------------"
echo "Extraction complete."
echo "Directory: $ARCHIVE_DIR"
echo "------------------------------------------------------------"
echo "To run the N-Dimensional Tensor simulation:"
echo ""
echo "1. cd $ARCHIVE_DIR"
echo "2. python3 run_simulation.py"
echo ""
```

```
echo "This will verify the transmission and reshaping of a 4D Hypercube."
echo "============================================================="
```