Building a Network Stack for xv6: E1000 Driver and UDP Protocol Implementation

Cameron VanderTuig, Islam Tayeb, Shaochen Wen, Zhuoxi Li
510 Operating Systems – Prof. Matthew Lentz

1 Introduction

The xv6 operating system is an educational OS used to teach fundamental concepts like process management, virtual memory, and file systems. However, it lacks networking capabilities, preventing students from learning how network protocols and device drivers work in practice. This project implements a complete UDP network stack for xv6, bridging this gap.

1.1 Motivation

Modern operating systems organize networking into layers: device drivers talk to hardware, protocol layers handle communication rules, and system calls let applications use the network. Understanding this requires building it yourself. Our work teaches how drivers coordinate with network cards using DMA, how protocol stacks parse packets, and how user programs access network services.

1.2 Contributions

We implement 2 main components:

E1000 Network Driver: Code that manages the Intel E1000 network card, handling packet transmission and reception through circular buffer rings.

UDP Protocol Stack: Code that processes incoming packets, routes them to the correct application based on port number, and provides system calls (`bind` and `recv`) for user programs.

Our implementation passes all MIT lab tests including basic send/receive, multi-port communication, DNS queries, and stress tests. Throughput testing shows the system handles over 4,000 packets per second with zero packet loss, with ring descriptor size having minimal impact on sustainable throughput.

2 Background

2.1 The Problem

xv6 includes skeleton network code in `kernel/e1000.c` and `kernel/net.c`, but the critical functions are empty stubs. The `e1000_transmit()` function cannot send packets to the network card, while `e1000_recv()` cannot retrieve packets from hardware. At the protocol layer, `ip_rx()` cannot process incoming UDP packets, and the system calls `bind()` and `recv()` provide no way for user programs to access the network. This leaves xv6 completely unable to communicate over a network.

2.2 Network Setup

We run xv6 in QEMU with an emulated E1000 network card. xv6 has IP address 10.0.2.15 and the host machine has 10.0.2.2 on a shared virtual network. Python test scripts on the host send and receive UDP packets to validate our implementation, allowing us to test the complete network stack from an external perspective.

2.3 The Mailbox Analogy

Understanding packet delivery in our system is easiest through a postal analogy. Think of UDP ports as mailboxes at an apartment building. When a process calls `bind(port)`, it's essentially claiming a mailbox and putting its name on it—the system now knows this process wants to receive mail (packets) sent to that address (port number). The `ip_rx()` function acts as the mailman, taking packets that have arrived at the building's main delivery area (the driver's receive ring) and sorting them into the correct mailboxes based on the destination port. Finally, `recv()` is the last-mile delivery—the process opens its mailbox, retrieves a letter, and brings it inside to read. Just like a real mailbox has limited space (our queues hold 16 packets), if too much mail arrives and the box fills up, new deliveries get dropped until space opens up.

2.4 Key Challenges

The E1000 uses Direct Memory Access with circular rings of descriptors, where the driver must manage these rings while tracking which buffers belong to hardware versus kernel and handling ring wrap-around correctly. Packets arrive asynchronously via DMA into the ring buffer descriptors, with the kernel being notified of this arrival via interrupts. User programs make system calls synchronously. Multiple programs might use different ports simultaneously, requiring careful locking to prevent race conditions. Each port needs a queue to store incoming packets, which we limit to 16 slots, meaning the system must handle queue overflow by dropping packets as UDP semantics allow, all while preventing memory leaks. Additionally, our implementation must handle the case where a process is killed before it finishes processing all queued packets—we must ensure proper cleanup to avoid resource leaks.

3 Design

3.1 Architecture

Our stack has four layers working together to enable network communication. At the bottom, the E1000 network card performs DMA on descriptor rings in memory. Above that, driver functions `e1000_transmit()` and `e1000_recv()` manage these rings and coordinate with hardware. The protocol layer's `ip_rx()` function parses packets and routes them to per-port queues based on destination port numbers. Finally, system calls `bind(port)` and `recv(port, ...)` provide the interface that user programs use to register ports and receive packets.

3.2 E1000 Driver

The E1000 has two circular rings: TX (transmit) and RX (receive). Each ring has descriptors that point to packet buffers. For transmission, the driver reads the tail register to find the next descriptor, checks if it's available by examining the "Descriptor Done" (DD) bit that hardware sets when finished, frees any old buffer at that slot (from the previous ring wrap-around), places the new packet in the descriptor with appropriate flags, and advances the tail register to notify hardware. The EOP flag marks end-of-packet (no fragmentation), while the RS flag tells hardware to report status when done. We use a separate transmit lock so only one process can modify the TX ring at a time.

For reception, the interrupt handler calls `e1000_recv()`, which processes all available packets in a loop to reduce context switch overhead. The function checks each descriptor's DD bit, and while set, extracts the packet buffer and length, passes it to the protocol layer, allocates a fresh buffer to keep the descriptor ready for new packets, clears status bits, and advances the tail register. The receive lock ensures the interrupt handler has exclusive access to the RX ring. Having separate locks for TX and RX allows concurrent sending and receiving, improving throughput when both operations occur simultaneously. It also prevents situations such as a receive-from-device interrupt occurring during transmission and causing a deadlock.

3.3 UDP Protocol Layer

We maintain an array of 32 ports, each containing a bound flag indicating if it's in use, the port number itself, a circular queue of 16 packets, and head, tail, and count variables for queue management. Each queued packet stores the payload data (up to 2048 bytes), the source IP address, and source port number, allowing applications to know where packets came from.

3.3.1 Packet Processing: ip_rx

When the driver delivers a packet, `ip_rx()` acts as the mailman, sorting incoming mail into the correct mailboxes:

```
ip_rx(buffer):
  parse ethernet/ip/udp headers
  extract dest_port, src_ip, src_port

  acquire(netlock)
  find port_entry for dest_port

  if port not bound or queue full:
    drop packet  // mailbox full
  else:
    copy payload to queue[tail]
    tail = (tail + 1) % QUEUESIZE
    count++
    wakeup(port)  // notify waiting process

  release(netlock)
  free(buffer)
```

Network byte order must be converted to host byte order using functions like `ntohs()` and `ntohl()`. After acquiring the netlock, the function searches the port array for a matching destination port. If the port isn't bound or the queue is full (16 packets), the packet is dropped as UDP allows. Otherwise, the payload is copied into the next queue slot along with source information, the tail pointer advances, and `wakeup()` is called to notify any process waiting for packets on this port.

3.3.2 Port Registration: sys_bind

Before receiving packets, a process must claim its mailbox by binding to a port:

```
sys_bind(port):
  acquire(netlock)

  if port already bound:
    release(netlock)
    return success  // already claimed

  find free port_entry
  if found:
    entry.bound = true
    entry.port = port
    entry.head = entry.tail = 0
    entry.count = 0

  release(netlock)
  return success or error
```

The function finds an empty slot in the port array and marks it bound, initializing the queue structures. This must be done before receiving packets on that port.

3.3.3 Packet Delivery: sys_recv

When a process is ready to read its mail, it retrieves packets from its mailbox:

```
sys_recv(port,  src, sport, buf, maxlen):
  acquire(netlock)

  find port_entry for port
  if not found:
    release(netlock)
    return error
```

```
while queue empty:
  sleep(port)  // wait for mailman

packet = queue[head]
head = (head + 1) % QUEUESIZE
count--

release(netlock)

copyout packet to user space
return packet length
```

The call waits using `sleep()` if the port's queue is empty, then dequeues the oldest packet (maintaining FIFO order), copies the data to user space using `copyout()`, and returns the packet length. The sleep/wakeup mechanism in xv6 automatically releases and reacquires the netlock, allowing other processes to continue enqueueing packets while one process waits.

3.4 Synchronization

Three locks prevent race conditions in our implementation. The TX lock ensures only one process can modify the transmit ring at a time, preventing corruption when multiple processes send packets concurrently. The RX lock gives the interrupt handler exclusive access to the receive ring during packet processing. The netlock protects the port array and all per-port queues, and is held during `bind()`, `recv()`, and `ip_rx()` calls. Processes can sleep while holding the netlock thanks to xv6's sleep/wakeup mechanism, which automatically releases the lock while sleeping and reacquires it when waking up. This prevents deadlock while still protecting shared data structures from concurrent access.

4 Implementation Details

4.1 Transmit

The transmit implementation checks the DD bit to determine if a descriptor is free for use, frees any previous buffer when wrapping around the ring to prevent memory leaks, and sets the EOP (end of packet) and RS (report status) flags to tell hardware this is a complete packet and to notify us when transmission completes. Hardware owns the buffer from this point until the next ring wrap-around, so the driver cannot touch it during this time.

4.2 Receive

The receive function loops until the DD bit is clear, indicating no more packets are available, which handles bursts of packets efficiently in a single interrupt. It allocates a new buffer immediately after processing each packet to keep the descriptor ready for incoming traffic. The RDT register points to the last processed descriptor, with hardware writing to the next one in sequence.

4.3 Protocol Processing

The protocol processing layer copies the payload to the queue rather than storing a pointer, allowing the original buffer to be freed immediately and simplifying memory management. Functions like `ntohs()` and `ntohl()` convert from network byte order (big-endian) to host byte order. Packets are dropped if the destination port isn't bound or the queue is full, which matches UDP semantics that allow packet loss. The `wakeup()` function notifies any processes blocked in `recv()`, allowing them to immediately process the newly arrived packet.

4.4 Process Termination Handling

When a process terminates while packets remain in its queue, our implementation ensures proper cleanup. The port entry remains valid until explicitly unbound, and queued packets are eventually freed when the queue is reused by a new binding. This prevents memory leaks even when processes exit unexpectedly before processing all their mail.

5 Evaluation

5.1 Correctness Testing

We use the MIT lab test suite (`nettest.c` and `nettest.py`) to validate implementation correctness. All tests described below were already implemented as part of the MIT 6.828 lab and our implementation successfully passes all of them.

Basic Transmission (txone): Tests the transmit path in isolation by having xv6 send a single packet that a Python script receives. This validates that `e1000_transmit()` correctly programs descriptors, sets appropriate flags, and notifies hardware without requiring the receive path to function.

Basic Reception (rx, rxburst): Test the receive path by sending numbered packets from the host. The `rx` test sends packets slowly (one per second) while `rxburst` sends them rapidly in bursts of 32. xv6 validates sequence numbers to confirm packets arrive in order. These tests validate that `e1000_recv()` extracts packets from the RX ring, `ip_rx()` parses headers correctly, and `recv()` delivers packets to user space in FIFO order.

Port Isolation (rx2): Validates that the port demultiplexing logic correctly separates traffic. The test sends interleaved packets to ports 2000 and 2001, then has xv6 call `recv()` on each port separately. Over 1000+ packets, the test confirms that packets sent to port 2000 only appear when calling `recv(2000)` and never contaminate port 2001's queue, demonstrating that the per-port queue management works correctly.

Bidirectional Communication (ping0, ping1, ping2): Test the complete network stack in both directions by having a Python script act as an echo server. The `ping0` test sends one packet and expects one reply, validating basic round-trip communication. The `ping1` test sends 20 packets in sequence, checking that all replies arrive in order and that FIFO ordering is maintained. The `ping2` test uses two different ports simultaneously, validating port isolation under bidirectional traffic where both sending and receiving occur concurrently.

Queue Overflow (ping3): Specifically tests queue management under stress by sending 257 packets rapidly to ports 2008 and 2010—far exceeding the 16-packet queue limit—bracketed by two packets on port 2009. This test confirms: (1) the queue limit is enforced and excess packets are dropped, (2) the system doesn't crash when queues overflow, (3) other ports continue working correctly during overflow conditions, and (4) no memory leaks occur when packets are dropped.

Real-World Protocol (dns): Validates protocol correctness by having xv6 query Google's DNS server (8.8.8.8) to resolve pdos.csail.mit.edu. This test ensures the stack handles actual protocol implementations with correct byte order conversion, proper header construction and parsing, and compatibility with external network services. The test confirms our implementation can interoperate with real-world network infrastructure beyond our test scripts.

Memory Stability (grade): Runs all previous tests in sequence while measuring free memory pages before and after. The test counts available pages using `sbrk()` and confirms that fewer than 32 pages difference exists after all tests complete. This validates that no buffer leaks occurred during the entire test suite—every allocated packet buffer was properly freed, even during error conditions and queue overflows.

All MIT lab tests pass successfully with the following results: single packet transmission and reception work correctly, multi-port isolation is verified over 1000+ packets with no cross-contamination, FIFO ordering is maintained consistently, queue overflow is handled correctly with exactly 16 packets queued per port and no system crashes, DNS queries succeed confirming protocol correctness and real-world interoperability, and memory remains stable after all tests with fewer than 32 pages difference indicating no significant leaks.

5.2 Throughput Testing

We tested the system's maximum sustainable throughput by sending 1000 packets at different rates and measuring packet loss. To efficiently find the breaking point, we used a binary search script: start by testing a very high rate (50,000 pkt/s) and a low rate (1,000 pkt/s). If the high rate fails and the low succeeds, test the midpoint. If the midpoint succeeds, the system can handle more—test higher. If it fails, the system is overloaded—test lower. This converges to the approximate maximum rate achieving 99% delivery. We tested two configurations: a 16-descriptor receive ring and a 64-descriptor ring.

Note that Table 1 shows an anomalous result at 4,061 pkt/s with 98.3% loss, likely due to a transient system issue during that specific test run. The binary search continued past this outlier, ultimately converging to 4,013 pkt/s where zero loss was consistently achieved. The 7,124 pkt/s test with only 1.7% loss suggests the true maximum sustainable rate lies somewhere between 4,013 and 7,124 pkt/s, but we report the conservative zero-loss threshold.

Table 1: 16-descriptor RX ring

Rate (pkt/s)	**Recv /1000**	**Loss (%)**	**Tput (pkt/s)**
25,500	371	62.9	73.0
13,249	629	37.1	100.9
7,124	983	1.7	188.9
4,061	17	98.3	2.8
2,530	1,000	0.0	638.6
3,295	1,000	0.0	675.6
3,678	1,000	0.0	730.3
3,869	1,000	0.0	732.9
3,965	1,000	0.0	728.7
4,013	1,000	0.0	723.3

Table 2: 64-descriptor RX ring

Rate (pkt/s)	**Recv /1000**	**Loss (%)**	**Tput (pkt/s)**
25,500	412	58.8	80.9
13,249	652	34.8	105.1
7,124	975	2.5	155.5
4,061	1,000	0.0	721.4
5,592	991	0.9	158.2
6,358	985	1.5	156.6
5,975	1,000	0.0	787.6
6,166	1,000	0.0	789.7
6,262	1,000	0.0	781.4
6,310	1,000	0.0	777.9

Tables 1 and 2 show results for both configurations. Each test sent 1000 packets at the specified rate and measured successful delivery. The 'Tput' column shows effective bidirectional throughput—the rate at which complete round-trips (send packet + receive echo response) are completed per second, which explains why it's lower than the one-way send rate.

At very high rates exceeding 25,000 packets per second, both configurations experience severe packet loss around 60%. Interrupt processing overhead cannot keep up with packet arrival—hardware fills the receive ring faster than the kernel can process descriptors. (Note the anomalous 4,061 pkt/s result in Table 1, which appears to be a test artifact rather than representative behavior given surrounding data points.)

The 16-descriptor ring achieves zero loss up to 4,013 packets per second, while the 64-descriptor ring maintains zero loss up to 6,310 packets per second. Despite having 300% more buffering capacity, the larger ring only provides 57% higher zero-loss throughput. More notably, both configurations achieve similar effective throughput in the 700-800 packets per second range when operating at their maximum zero-loss send rates.

This suggests the bottleneck is not hardware buffering but system call and kernel processing overhead. Each packet requires acquiring locks, dequeuing from the circular buffer, calling `copyout()` to cross the kernel-user boundary, validating payload, and echoing back. These software operations dominate per-packet processing time far more than waiting in hardware queues, explaining why additional descriptor capacity provides only modest improvement.

5.3 Latency Testing

We tested the roundtrip latency from host to xv6 back to host at different throughputs. To do this, we have multiple throughput buckets and test ping time originating at the host. For each throughput bucket (1 packets per second, 10 packets per second, etc.), we send throughput*5 packets and wait for the reciprocal each iteration, meaning each bucket tests for 5 seconds total. Results are shown in Table 3.

Table 3: Synchronous Latency Results

Throughput (msg/s)	**Average (ms)**	**Median (ms)**	**P95 (ms)**	**P99 (ms)**	**Min (ms)**	**Max (ms)**	**Samples**
1	1.581	1.646	1.932	1.932	0.982	1.932	5
10	1.813	1.796	2.182	3.169	1.359	3.169	50
100	1.720	1.547	2.817	4.899	0.713	23.017	500
1000	1.225	1.176	1.472	2.854	0.413	39.223	5000

These results are surprising. Average time decreases with increased throughput while the max latency widens. We theorize that this may be due to the sending and receiving running ”hot”. For example, in our implementation of the network driver, when receiving from the device, we check every descriptor that is available for processing, which may be many in the same interrupt if the throughput is high enough. Instances like this may explain the lower latency at higher throughputs.

6 Conclusion

We successfully implemented UDP networking for xv6, including the E1000 driver and protocol processing. The implementation passes all MIT lab tests, handles over 4,000 packets per second with zero loss, and maintains stable memory usage.

Future extensions could add TCP support, socket APIs, or network utilities like ping and telnet. The current implementation provides a solid foundation for understanding network operating system internals.