

# cmsc 417 programming assignment #3

March 4, 2008

Goal: Implement unreliable rate-controlled message delivery to neighbors.

## 1 Deadline

DUE: March 14. late deadline will be Monday, March 17; you have only one late deadline to play with.

## 2 Context

This project extends the previous project by providing rate-controlled message transmission. We'll add a command called `sendmsg`<sup>1</sup> that will inject a message into the network *when* doing so will not exceed a rate limit. The rate-control requirement ensures that messages from a host do not swamp its neighbors or the network. Because the service runs over UDP (over IP), without any retransmission, it remains unreliable.

The rate control is the so-called “leaky bucket” algorithm. Water (packets) pours into a leaky bucket (queue). When the bucket has water, it leaks steadily (sends at fixed rate). If the incoming water flow is so fast that it fills the bucket (queue is filled), water spills out (packets are dropped).

## 3 Service specification

The `sendmsg` service:

- Takes `dst-address`, the neighbor's logical address (the 32 bit source address in the packet header, not the IP address), and `msg`, a message to send, as parameters. Consider: `sscanf("sendmsg %u %s", ...)`, which might work.
- Limits the rate of sending to 10 packets (of maximum size 1000 bytes each) per second to each neighbor (that is, per second per neighbor)
- Uses a per-neighbor output queue of maximum size 10 packets (irrespective of packet size) to limit the sending rate

When `sendmsg` is invoked, it does the following in sequence:

- Look up the IP address and UDP port of the neighbor's unicast socket in the neighbor table. (Don't send messages to the multicast address!)
- Check if last packet was sent 0.1 seconds ago or earlier:
  - If so, send the message immediately using UDP `sendto`
  - If not, enqueue the message in the output queue subject to maximum queue size (10).

---

<sup>1</sup>Think of `sendmsg` as a wrapper over UDP `sendto`. This is an example of how services are composed over layers, i.e., a higher protocol layer uses a lower layer's service specification to provide something more.

- If the queue is full, drop packet
- Ensures that the queue, if not empty, is drained by one packet every 0.1 seconds.

## 4 Implementation requirements

In addition to the requirements of P2, P3 should:

- Implement a command from STDIN with signature `sendmsg dst-address message` providing the semantics of the `sendmsg` service.
- Handle queue overflow by printing “ERROR:NOBUFF”. Do not block the `sendmsg` command waiting for the queue to drain.
- Handle unknown or dead neighbor by printing “ERROR:NOROUTE”
- Not leak memory
- Print received (version 1, protocol 1) messages on the unicast socket with `printf("U: %.*s", ...)`.

## 5 Hints

- Figure out what events will be required to drain the buffer (queue) periodically and use your event queue appropriately.
- Be careful when **freeing** packets; you want to free them, but only when you actually can.
- You should be able to test by sending to yourself.
- `sendmsg` is unrelated to the system call of the same name. You may use that one if you feel so inclined, but there’s no reason to.
- Multicast transmissions *may* also be rate-limited if you like, but the only transmission that will remain in multicast is the periodic hello, which generated on
- Receiving a unicast message from a neighbor *may refresh* the neighbor’s entry in the neighbor table. It should not create a new entry, because eventually, we’ll have forwarding, and the source address of a forwarded message is not a neighbor.

## 6 Objectives

At the end of this assignment, you should understand:

- maintaining queues of packets without coredumps :)
- scheduling events to manipulate queues of packets.
- simple rate control.

## 7 Free rein

You may complete this assignment in C or Ruby. Ruby is an object-oriented scripting language based on Smalltalk. An on-line copy of the “Pickaxe” book on Ruby is at <http://www.rubycentral.com/pickaxe/>. The rules for a ruby implementation remain the same (no threads, no alarms).

To process packets in ruby requires using pack and unpack functions that can convert between packets as strings and arrays of constituent header values.

Later assignments may also be completed in C or Ruby.