Project 3: Reliable Communication

Due: Thursday 12/15 by 5pm.

For your questions: `ecalp at cs dot wisc dot edu`

News:

- **12/12:** FAQ is updated. Make sure to check it before sending an email!
- **12/07:** Example .txt files are now available to make sure that we are on the same page on their format.
- **12/07:** Explanation added for `recv_timeout` (in bold).
- **12/04:** Check out the new additional task!
- **11/22:** Project 3 is now available

Overview

In your final assignment you are going to build a reliable communication library in Switchyard that will consist of 3 agents. At a high level, a **blaster** will send data packets to a **blastee** through a **middlebox**. As you should all know by know, IP only offers a best-effort service of delivering packets between hosts. This means all sorts of bad things can happen to your packets once they are in the network: they can get lost, arbitrarily delayed or duplicated. Your communication library will provide additional delivery guarantees by implementing some basic mechanisms at the blaster and blastee. Let's move on to the details.

Details

Your reliable communication library will implement the following features to provide additional guarantees:

1. ACK mechanism on blastee for each successfully received packet
2. A fixed-size sliding window on blaster
3. Coarse timeouts on blaster to resend non-ACK'd packets

Let's further break down the features of each agent.

Middlebox

Even though I call it a middlebox, in practice this will be a very basic version of the router you implemented in P2. Your middlebox will have two ports each with a single connection: one to the blaster and one for the blastee. You will do the same packet header modifications(i.e layer 2) as in P2. However instead of making ARP requests you will hard code the IP-MAC mappings into the middlebox code. This means, if the middlebox receives a packet from its eth0 interface(= from blaster), it will forward it out from eth1(= to blastee) and vice versa. This basic assumption also obviates the need to do forwarding table lookups. Regardless of the source IP address, just forward the packet from the other interface.

So far so good. Now comes the fun part of the middlebox! Besides a very dumb forwarding mechanism, your middlebox will also be in charge of probabilistically dropping packets to simulate all the evil things that can happen in a real network. Packet drops will only happen in one direction, from blaster to blastee (i.e do not drop ACKs).

Blastee

Blastee will receive data packets from the blaster and immediately ACK them. Blastee will extract the sequence number information from the received data packet and create an ACK packet with the same sequence number. Unlike TCP where sequence numbers work on bytes, your implementation will use sequence numbers at packet granularity.

Blaster

Blaster will send/receive variable size IP packets/ACKs to/from the blastee. As mentioned above, it will implement a fixed-size sender window (SW) at packet granularity and coarse timeouts. In order to clarify how SW will work, let's define two variables LHS and RHS(both always >= 1), where LHS and RHS (correspond to sequence numbers of 2 sent packets that have not been necessarily ACKd yet) indicate the current borders of the SW such that it always satisfies the following:

C1: $RHS - LHS + 1 \leq SW$

SW effectively puts a limit on the maximum number of unACKd packets that can be in-flight between the blaster and blastee. Logic of changing the RHS is simple, as the blaster sends packets it will increment the RHS value without violating the previous condition. However, changing the LHS value is more tricky. LHS tells us the packet with the lowest sequence number S_i such that:

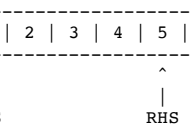
C2: Every packet with sequence number $S_j < S_i$ has been successfully ACKd.

Let's look at the following example to better understand this. Numbers in the boxes indicate the sequence number associated with each packet:

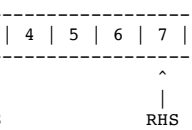
Suppose `SW=5`. Initially `LHS=1 and RHS=1`

```
 ---------------------
 | 1 | 2 | 3 | 4 | 5 |
 ---------------------
   ^
   |
  LHS
  RHS
```

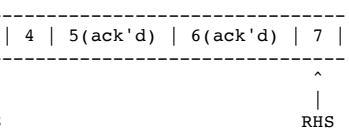
Based on the explanations above, blaster will start sending packets and it will keep incrementing the RHS value. After sending the first 5 packets and not receiving any ACKs, the SW will look like this:



Note that we can't move the RHS any further otherwise we will violate C1. This also means that blaster can't send any new packet to the blastee until it starts receiving ACKs. Let's assume that ACKs for packets #1 and #2 arrive at the blaster. In this case LHS will point at #3 and therefore we can move the RHS to 7.



Now let's assume that the middlebox dropped packets #3 and #4, which means the blastee won't be able to ACK them. After a while, ACKs for #5 and #6 arrive at the blaster.

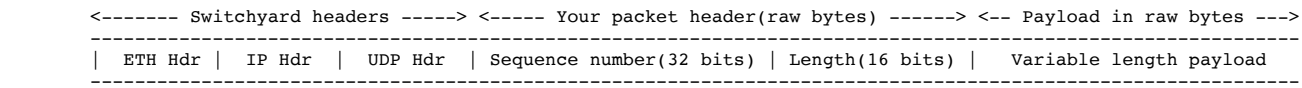


Notice that even though the blaster received some ACKs for its outstanding packets, since C2 is not satisfied LHS can't be moved forward which also prevents RHS from moving forward (to not violate C1). As you can see unless we implement an additional mechanism, blaster will be stuck in this position forever. This is where the **coarse timeouts** come into play. Whenever LHS gets stuck at a position for longer than a certain amount of time, blaster will time out and retransmit every packet in the current window that hasn't been ACKd yet. So in the previous example if LHS doesn't change for the duration of the timeout period and only packets #5 and #6 are acknowledged in the meantime, blaster will retransmit #3, #4 and #7 upon timing out. Keep in mind that some weird things can happen in this process: 1) blaster can receive an ACK for the original tranmission of a packet after retranmsitting it or 2) blaster can receive duplicate ACKs. For this project you don't need to worry about these and just keep track of whether a packet is ACKd or not.

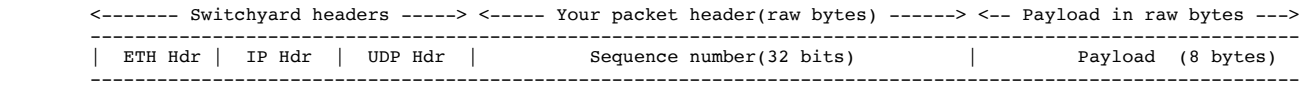
Packet format

The packets will have 3 headers: `Ethernet`, `IPv4`, `UDP`. It's obvious why you will have the first 2 headers. UDP header will serve as a placeholder to prevent Switchyard from complaining. You can read about the parameters of UDP header here. You can assign arbitrary values for the port values as you won't be using them. You will append your packet to this sequence of packets. I suggest you use the `RawPacketContents` header in Switchyard. It is just a packet header class that wraps a set of raw bytes. You can find some information about it on the same web site. You can also take a look at the source code to understand how it works. Or better, you can just test it on your own!

Here is how your data packet will look like:



Here is how your ACK packet will look like:



Notice that the ACK packet will have a fixed size payload (8 bytes). You will populate these bytes from the first 8 bytes of the variable length payload of the blaster's packet that you received at the blastee. If the blaster's packet has a payload with less than 8 bytes, just pad the payload in ACK as you need.

You will need to encode the sequence number and/or length information in to your packets, which will be in raw byte format. Encoding should use **big-endian** format! Python has built-in library calls to achieve this with minimum pain.

Printing stats

Once the blaster finishes transmission (which happens upon successfully receiving an ACK for every packet it sent to the blastee -- equals to `num`), it is going to print some statistics about the transmission:

- ***Total TX time (in seconds)***: Time between the first packet sent and last packet ACKd
- ***Number of reTX***: Number of retransmitted packets, this doesn't include the first transmission of a packet. Also if the same packet is retransmitted more than once, all of them will count.
- ***Number of coarse TOs***: Number of coarse timeouts
- ***Throughput (Bps)***: You will obtain this value by dividing the total # of sent bytes(from blaster to blastee) by total TX time. This will include all the retransmissions as well! When calculating the bytes, only consider the length of the variable length payload!
- ***Goodput (Bps)***: You will obtain this value by dividing the total # of sent bytes(from blaster to blastee) by total TX time. However, this will **NOT** include the bytes sent due to retransmissions! When calculating the bytes, only consider the length of the variable length payload!

Running your code

Instead of running your implementations with test scenarios, you will be running the agents in Mininet. I am providing you with a topology file (`start_mininet.py`) in Mininet. I added comments so please read them to understand the setup. Please do not change the addresses(IP and MAC) or node/link setup. I will be using the same topology file when testing your code (although I reserve the right to use different delay values). To spin up your agents in Mininet:

1. Open up a terminal and type the following command. This will get Mininet started and build your topology:

   ```
   $  sudo python start_mininet.py
   ```

2. Open up a xterm on each agent:

   ```
   mininet> xterm blaster
   mininet> xterm blastee
   mininet> xterm middlebox
   ```

3. Start your agents:

   ```
   blaster# ./switchyard/srpy.py blaster.py
   blastee# ./switchyard/srpy.py blastee.py
   middlebox# ./switchyard/srpy.py middlebox.py
   ```

You will need to specify some parameters for each agent when running them. To my knowledge, it is not possible to pass custom parameters to Switchyard (probably why you assumed there was a forwarding_table.txt file in your working directory in last project as well). However, this is not going to keep us from passing parameters to our agents. Just like in the previous project you will assume that there will be 3 files in your working directory: **blaster_params.txt**, **blastee_params.txt** and **middlebox_params.txt**.

blaster_params.txt will contain the following line:

```
-b <blastee_IP> -n <num> -l <length> -w <sender_window> -t <timeout> -r <recv_timeout>
```

- *blastee_IP*: IP address of the blastee. This value has to match the IP address value in the `start_mininet.py` file
- *num*: Number of packets to be sent by the blaster
- *length*: Length of the variable payload part of your packet in bytes, $0 \leq length \leq 65535$
- *sender_window*: Size of the sender window in packets
- *timeout*: Coarse timeout value in milliseconds
- *recv_timeout*: `recv_packet` timeout value in **milliseconds**. Blaster will block on `recv_packet` for at most *recv_timeout*. This will be a pseudo-rate controller for the blaster

blastee_params.txt will contain the following line:

```
-b <blaster_IP> -n <num>
```

- *blaster_IP*: IP address of the blaster. This value has to match the IP address value in the `start_mininet.py` file
- *num*: Number of packets to be sent by the blaster

middlebox_params.txt will contain the following line:

```
-d <drop_rate>
```

- *drop_rate*: Percentage of packets (non-ACK) that your middlebox is going to drop, $0 \leq drop\_rate \leq 1$

Starter files

You can find them here.

Testing your code

Good news: you aren't going to write test cases to test your implementation. Bad news: you still need to test your code. In order to make sure that your blaster, blastee and middlebox function correctly you will have to use Mininet. The process is explained above.

Handing it in

For submitting your work, you should put the following files into one partner's handin directory. Handin directory is `~cs640-1/handin/login/p3` where `login` is your CS login.

- **blastee.py**: Your blastee implementation
- **blaster.py**: Your blaster implementation
- **middlebox.py**: Your middlebox implementation
- **README.txt**: This file will contain a line for each student in your team: [name_of_student][single whitespace][10-digit UWID]

IMPORTANT: The file names in your handin directory has to **exactly match the file names above**. Otherwise, you will lose points!

Grading

50% of your grade will be based on successfully implementing the functional components of the reliable communication library to create the target functionalities. The remaining 50% will be based on the success of your implementation over a range of conditions.

FAQ

1. **Q:** How can the blaster, blastee and middlebox know the MAC and IP addresses of each other's ports?

   A: You will use the addresses that are specified in start_mininet.py file. You can hard code the addresses in your code.

2. **Q:** Where do the .txt files, which are used to pass parameters to your agents, come from?

   A: You will create these .txt files with parameters of your choice.

3. **Q:** How does the dropping of packets work?

   A: You will use the `drop_rate` parameter at your middlebox for deciding whether you should drop the packet or not. You can use the RNG function in Python to obtain a value (between 0 and 1) and then compare it against the `drop_rate` to decide what to do with the packet.

4. **Q:** Are we supposed to maintain a timer for each outstanding packet?

   A: Coarse timeout applies to the whole sender window, therefore you don't need to maintain a separate timer for each outstanding packet. You will reset your timer whenever you move LHS. If the LHS value doesn't change for the duration of `timeout` then you will retransmit every non-ACKd packet in that window.

5. **Q:** Can you give us an example for the coarse timeouts?

   A: Let's say the `sender_window` is 4 and the `timeout` is 1 second. Initially, blaster sends packets 1,2,3 and 4. Blaster receives the ACK for 2 at 300 ms into the coarse timeout (can't move LHS or RHS since packet 1 not ACKd yet). Then it receives the ACK for 1 at 500 ms. Now, blaster will move its LHS to point at 3, reset its timer (since LHS changed), send out packets 5 and 6, move RHS to 6 (`RHS(6) - LHS(3) + 1 ≤ SW(4)`). Even though packet 3 and 4 were sent in the previous sender window, they will not carry their timer values to the new window. Now let's say packets 4 and 5 are immediately ACKd (around ~500 ms since the initialization of the system) by the blastee and the blaster doesn't receive the ACK for packet 3 before the coarse timeout occurs. Now the system time will roughly be at 1.5 seconds(as timeout = 1 second) and since the coarse timeout occurred, blaster will retransmit packets 3 and 6.

6. **Q:** I get the following error when I try testing my switch implementation in Mininet: ***NameError: name 'devname' is not defined (switchy_real.py, line 261 in send_packet())***. How can I fix it?

   A: Please go ahead and pull down the updated version of Switchyard from the GitHub repository. This issue was fixed during P1.

7. **Q:** Can I assume that the .txt files will have the parameters in the same order as illustrated above?

   A: Yes, you can safely assume that.

8. **Q:** Can `recv_timeout` be greater than `timeout`?

   A: No, you can safely assume that `recv_timeout` < `timeout`. If you think about it would be weird to have it the other way around as it would prevent the blaster from timing out in a timely manner.

9. **Q:** Can we send multiple packets from the blaster at each recv_timeout loop? Or do we have to send a single packet at each loop (because recv_timeout is a pseudo-rate controller)? How about the retranmissions?

   A: You should be sending a single packet per each `recv_timeout` loop. As I mentioned in the specification, `recv_timeout` will work as a pseudo-rate controller. Recall that you pass `recv_timeout` to the recv_packet function in Switchyard and this determines how long this function will block before timing out. I use the word pseudo on purpose because your blaster can receive ACKs fast enough (and without many losses) that the recv_packet call would never time out and your blaster can send packets at a faster rate. You should follow the same logic in retransmissions.