URP Report

Yoel Yoffe (z5589593)

Overview and Project Layout

The URP implementation is written in Python. The project structure is simple: `sender.py` and `receiver.py` are the main two files, and both inherit from `common.py`. The sender and receiver can be run as indicated in the assignment specification. The receiver generates a log in `receiver_log.txt`, and the sender generates a log in `sender_log.txt`. The file is transferred over UDP sockets using a sliding-window mechanism.

Protocol Implementation

Overall Design

The overall program has been designed using an object oriented programming style. The sender and receiver are designed as classes to allow them to store internal state. The main external-facing method for these classes is `.run()`, which causes the sender or receiver to go through all states in its state transition diagram. Threading is used to control retransmission timeouts as well as the timed wait period of the receiver.

The sender and receiver rely on helper classes and functions included in `common.py`. One of these is a checksum class; this will be discussed later in the report. Several modulo arithmetic functions have also been implemented to allow for clean handling of sequence number wraparound. These include `wrap_add()`, `wrap_sub()` and `wrap_cmp()`. The functions `wrap_add()` and `wrap_sub()` have standard implementations. The function `wrap_cmp()` assumes that if the two integers it is comparing differ by more than half the size of the sequence number space, then a wraparound of the smaller integer has occurred. This assumption is reasonable, considering sequence numbers increase by much less than half of the sequence number space each time. Since the window size must be less than half the sequence number space, there is never a case where the `wrap_cmp` logic can lead to issues. Finally, `common.py` includes a `Segment` class. This is an abstraction of a URP segment which provides numerous convenience features. The constructor can be used to specify various URP segment properties like payload, sequence number and flags. The `.encode()` method is used to encode these fields as a byte array. The checksum is automatically generated when a segment is encoded. The input to the checksum is the entire encoded segment (with zeroes in place of the checksum field). The `.create()` factory method provides a simple interface for creating segments of a certain type. The `.decode()` static method parses a byte array into a segment. If the segment header is corrupted it does not attempt to create the segment; if the payload is corrupted the `.decode()` method still creates the segment, but warns the caller about corruption via a return value. A case where the segment header is corrupted should never occur with our corruption scheme, but is useful for generality. Thus, the Segment class is an effective abstraction of the details of segment encoding, decoding, and checksum verification.

Packet Loss and Corruption (PLC) Module

The PLC module has been implemented as a class in `sender.py`. It has two main interface methods, .send() and .recv(), which act as a black-box abstraction of an unreliable channel.

The `.send()` method first preforms a Bernoulli trial to see if it should drop the segment. If the segment should be drop, it writes a message to the log and updates its statistics. Otherwise, it performs another Bernoulli trial (independent of the first) to check if it should corrupt the segment. If the trial is successful, it flips one bit in the segment payload, updates its statistics, and writes to the log file. Finally, it sends the data through the socket. The entire method is enclosed in a lock, since it may be called concurrently from the retransmission thread and the main thread. The `.recv()` method functions analogously, first checking whether it should drop the received segment, and then checking whether it should corrupt the received segment. Logs are made as appropriate.

There is no communication between the PLC module and the outside world. The sender and receiver detect corruption and packet drops through their own mechanisms (timeouts and checksums); they do not use any information provided by the PLC module.

Sender
The sender stores key fields inside a state control block, which is defined as a nested class. A key field in the state control block is a buffer for storing unacknowledged segments, which is implemented as a double ended queue. The double ended queue allows for easy removal of newly acknowledged segment from the front of the queu, and insertion of newly sent segments at the back of the queue. The state control block is also lockable to allow multiple threads to perform updates to it without races.

When the sender is run via the `.run()` method it immediately sends a SYN segment and enters the syn_sent state. The SYN segment is sent via a special stop-wait exchange subroutine, which waits for a valid ack before proceeding further. Once the ack is received the connection is established and the sender enters the established state. It then enters a data sending loop, where it transmits all segments inside its send window and waits for acks.

Transmitting segments inside the send window involves several steps. First, the number of bytes to transmit is calculated by taking the maximum between the window size and the MSS. Data is read from the input file just before it is transmitted (this means that the whole input file is never stored in memory at once). If the entire file has been read, then the state of the sender is changed to closing and the window transmission subroutine terminates early. Otherwise, a segment is created and sent through the socket.

Receiving an ack also involves several steps of control flow. If the ack is below the current base of the send window it is discarded. If the ack is instead equal to the base of the send window it is a duplicate ack, and the dup ack counter is incremented. If three duplicate acks are received a fast retransmission is triggered. If the ack is above the send window base some new segments have been received; we pop these segments from the unacked queue. We then check if the unacked queue is empty; if it is we cancel any outstanding retransmission timers. We also check whether this is an ack for the last data segment, since then we need to transition to the fin_wait state. Finally, we implement segment trimming. If an ack is in the middle of an unacked segment, the segment's data is trimmed to start at the first unacknowledged byte. This should never be used in our URP protocol, but is useful for generality.

A design decision was made to have the window transmission and ack handling subroutines both run on the main thread, with ack handling following window transmission. Although these are separate events, their strong coupling means it is not useful to separate them into multiple threads. Specifically, the send window can only move forward if an ack is received. It therefore makes sense to wait for an ack to actually be processed before running the window transmission subroutine again. Running the two in separate threads would require complex event signalling mechanisms, which are unnecessary, and would bring no benefit in terms of execution speed or speed at which data is pipelined through the socket.

Once all data has been transmitted, the sender performs another stop-wait exchange to send the FIN segment. Once the corresponding ack has been received it enter the closed state and terminates.

Whenever the unacked queue is nonempty, a retransmission timer runs on a separate thread. When this timer fires the earliest unsent segment is resent. Locks are implemented to prevent conflicts between the main thread and the retransmission thread.

Receiver
Similarly to the sender, the key fields of the receiver are stored in its state control block, which is implemented as a nested class. The most important field in the state control block is the buffer, which is implemented as a double ended queue.

When the receiver is run (via the .run() method), it immediately changes its state to listen. It then handles the stop-wait SYN-ACK exchange with the sender. When the first SYN is received, it enters the established state and sends back an ack. Subsequent SYN segments may be received if the ack was dropped or corrupted; in this case the receiver just resends the ack.

Once the SYN-ACK exchange has been completed, the receiver enters a data-receiving loop. On the receipt of a data segment the .process_data_segment() method is called. This method drops any packets beneath or above the receive window. A packet beneath the receive window is a duplicate packet; a packet above the receive window indicates that the sender's window is bigger than ours. If the packet is within the window, the receiver runs a check for whether it is the next expected packet. If it is, it places it as the first item in the buffer and pops all in-order packets from the buffer, flushing them to the output file. If the packet has arrived out of order the receiver instead places it at the correct location in the buffer. Note that the buffer size can never exceed the receive window size, as is expected by the specification. After processing the data packet, the receiver sends back an ack with the sequence number corresponding to the next packet it expects. The data receiving loop terminates when a FIN packet is received.

On receiving a FIN the receiver sends back an ack and enters the time_wait state. It sets a timer for twice the maximum segment lifetime. When this timer fires, the state of the receiver will be changed to closed. Until then, the receiver polls the socket in a loop (using nonblocking socket IO). If further FIN packets are received it sends back an ack and restarts the timer. When the closed state is entered the receiver terminates.

When designing the receiver the decision was made to use a double ended queue as the buffer data structure. This allows for incoming in-order segments to be quickly inserted at the front of the buffer, and for items to be quickly popped from the front of the buffer. Searching for a place to insert an out-of-order segment is done via a linear scan: since window sizes encompass around 30 segments or below there is no need to use other mechanisms like binary searches or red-black trees. A linear scan is simple and has low constant factors.

Checksum Mechanism
The checksum mechanism used is a 16-bit cyclic redundancy check. The code can be found in the CRC16 class of `common.py`. The cyclic redundancy check involves taking a polynomial of degree 17 with all coefficients either 0 or 1 and performing a polynomial division with the polynomial that is represented by the data bits. The degree-16 remainder polynomial, represented as a bit pattern, is the checksum value. Each division step is accomplished by taking bitwise XORs.

My implementation uses the polynomial 0x16F63, which was listed as a common CRC-16 polynomial in online sources (e.g. https://users.ece.cmu.edu/~koopman/crc/crc16.html) . Testing using 10000 randomly generated byte strings of length from 1 to 10000 demonstrated a 100% accuracy rate, with all corrupted packets being detected as such and all uncorrupted packets passing the checksum.

The core of the implementation lies in the _get_remainder() function of the CRC16 class. The function first adds a 16-bit trailer of 0s to the data. While there are 1s remaining in the data, the function takes a XOR with the specified polynomial in order to clear the ones, simulating a polynomial division. Once all 1s are cleared, the content of the 16-bit trailer is returned as the checksum.

A common speedup to the CRC checksum is a CRC table, which allows one byte to be XORed at a time instead of one bit. This was deemed unnecessary for the implementation, as the speed gains for our small segment sizes of 1000 bytes would not be noticeable.

Errors
There have been no errors found in the implementation during testing. The implementation has been used to transfer large files (6MB), as well as files under adverse conditions (transfer of rfc793.txt succeeds when flp,rlp,fcp,rcp are all 0.9, though it takes a long time). Sliding windows up to 30 000 have been tested. The sliding window behaviour can be demonstrated by setting a large window size (e.g. 8000), and setting rcp to 0.9 while all other probabilities are 0. If this is done the general pattern in the log file is that the entire window is sent, then an ack for the entire window eventually arrives, causing another full window to be sent, with the process repeating until termination.