Deev Patel
CPE 400 – Fall 2020
12/07/20
Final Project Code Documentation

ECR Protocol Simulation Code Documentation

Overview

The simulation code is written in Python with the help of the arcade library (https://arcade.academy/). The program takes two files to run: a network structure file and a simulation packets file. Once run, it opens a simulation window that can be controlled by the user.

Overview: Network Structure File

The network structure file contains the nodes and bidirectional links in the simulated network. Each node has a name, initial battery level, and position in the world. The world is 2-D with position (x=0, y=0) on the bottom left.

Overview: Simulation Packets File

The simulation packets file contains the packet streams that will be routed in the simulation. Each packet stream models an application layer request at some specific simulation time-step to send some number of packets from a source to a destination. The number of packets to be sent can be infinite (in which case the packets will be sent until there are no available routes due to nodes going offline).

Overview: Simulation Setup

Once simulation has started, one can see the simulation time in the top right. Each simulation step time-step is 1 millisecond. So, a simulation time of 1000 means one second has passed in the simulation. In the simulation, we assume that it takes 1 simulation time-step for a packet to go across each link. We assume no other delays. At each time-step, the sent packets will be marked as inflight and processed at the next time-step on the opposite side of the link. The simulation environment displays possible user input options at the top right. The simulation environment will also log everything to the terminal and various logs. Screenshots of possible environments and outputs can be seen in the associated report.

Running

Running: Installation

The program was tested with Python 3.8 on both Windows 10 and Ubuntu 20.04 LTS. It may or may not work with older versions of python. Instructions on installing Python 3.8 can be found on the official Python website (https://www.python.org/downloads/release/python-380/). Once python is installed, one needs the arcade library, which can be installed via `pip3 install arcade`. Once installed, one should be able to execute `main.py`.

Command Line Flags

- `--network_file`: File defining network nodes and links.
- `--packets_file`: File defining what application layer packets need to be sent during simulation.
- `--log_file_full`: Log file for full log text.
  - default value: `logs/log_full.txt`

- `--log_file_packets`: Log file for simulation packets defined in the packets file.
    - default value: `logs/log_packets.txt`
- `--log_file_errors`: Log file for errors that were handled by ECR's inbuilt error handling.
    - default value: `logs/log_errors.txt`
- `--log_file_performance`: Log file for packet performance. Describes how the required packets were routed at different times.
    - default value: `logs/log_performance.txt`
- `--log_file_energy`: Log file for energy level of network. Holds the average energy per node at each simulation time-step.
    - default value: `logs/log_energy.txt`

Running: Simulation 01

`python3.8 main.py --network_file config_files/sim01_nodes.txt --packets_file config_files/sim01_packets.txt`

Running: Simulation 02

`python3.8 main.py --network_file config_files/sim02_nodes.txt --packets_file config_files/sim02_packets.txt`

Running: Simulation 03

`python3.8 main.py --network_file config_files/sim03_nodes.txt --packets_file config_files/sim03_packets.txt`

Running: Simulation 04

`python3.8 main.py --network_file config_files/sim04_nodes.txt --packets_file config_files/sim04_packets.txt`

Implementation Summary

Implementation Summary: `main.py`

Contains the main entry function of the program. This file also defines the various command line arguments. The main function does four things: (1) create the simulation through a `NetworkSimulation` object, (2) load the network structure and packet schedule into the simulation, (3) run the simulation, and (4) print the overall results.

Implementation Summary: `Constants.py`

This file contains various constants for the program. Notably (at the top), the file contains constants dictating the world, screen, and text sizes. If a user wants to reduce the size of the simulation window, this is where it can be done. At the bottom of the file are constants specific to the ECR protocol. The details on many of the constants can be found in the associated report.

Implementation Summary: `NetworkLogger.py`

This file defines a `NetworkLogger` class. The class is used for logging purposes and provides an easy interface to either print data to the terminal or log data to the various specific log files. The class handles all log file creation and cleanup. The class also keeps track of the number of errors logged.

Implementation Summary: `PacketTypes.py`

This class defines the various message types needed for the ECR protocol. See the associated paper for how each message is structured and used. The message types are:

- `ERC_RD`: Route Discovery Message
- `ERC_RR`: Route Response Message
- `ERC_RP`: Route Packet Message
- `ERC_RU`: Route Update Message
- `ERC_RE`: Route Error Message

The file also defines a `Packet` class that wraps the message types. The `Packet` class is designed to allow for a common interface to all the message type and provides a uniform way to simulate in-flight packets. In addition to the underlying `msg`, a `Packet` has a `current_hop` and `next_hop` indicating where it is at the current time-step and where it will be at the next time-step. So, as a message is routed through a network, only the `Packet` wrapper's attributes will be modified.

Implementation Summary: `Helper.py`

This file defines various helper functions to make things easier. It provides functions to create and maintain unique link and route names. These unique names are used to create dictionary mappings. The file also defines a function to parse the network input file, `load_nodes()`, and another function to parse the network packets file, `load_simulation_packets()`.

Implementation Summary: `NetworkNode.py`

This file defines a `NetworkNode` class that represents a ECR compatible node or router in the network.

NetworkNode: Member Variables

Each node models its battery level and holds various attributes (like an RMT and an exponential moving average of number of packets sent) to be able to implement the ECR protocol. Each node also contains various dictionaries and maps holding a history of how packets were routed/received. These are used for generating protocol metrics and not strictly needed for the protocol itself. See the associated code comments for details on the various specific member variables.

NetworkNode: Class Functions

The class has a variety of functions that implement the ECR routing protocol on a nodal level. See the code for more detailed comments. The notable functions include:

- `progress()`: Progresses the node to the next simulation time-step. Applies (Eq. 1) and (Eq. 2) to compute the nodal last-alive-time. Also applies (Eq. 4) to update any RMT entries based on the update to the nodal last-alive-time. Note that this function needs to be called sequentially at each time-step.
- `update_or_create_rmt_entry()`: Updates or creates an RMT entry based on (Eq. 3). Called whenever the node gets a link maintenance communication from a neighbor. This function is also called to create and update RMT entries based on RR/RU/RP messages the node is receiving or forwarding.
- `get_best_route()`: Searches the RMT route cache and returns the best possible route. The search criteria are defined in the associated paper.
- `cleanup_dead_neighbor()`: Removes RMT entries associated with a neighbors that has not sent a link maintenance message for an extended period of time (ie: neighbor is offline).
- `generate_route_discover_packets()`: Generates a list of RD messages that should be sent in order to discover a route to a specific destination. This function automatically handles timeouts and waiting for already sent RD messages (ex: if RD messages for a specific route where sent recently, this function will not generate new RD messages).
- `attempt_to_send_packet()`: Attempts to send an application layer packet to the given destination. If a route is present in the RMT, this will return a RP message being sent to the destination. If a route is not known, this will return RD packets needed to search for a route. This function will also check for any timeout errors.
- `handle_packet()`: Handles a packet. This function will handle each ECR packet type appropriately. It will forward any necessary packet along and, if needed, create new response packets. See the associated report for details on how each packet type works.

Implementation Summary: `NetworkSimulation.py`

This file defines a `NetworkSimulation` class that holds the simulation world and nodes. The class is responsible for running the simulation. This involves everything from handling user inputs, to keeping track of in-flight packets, to scheduling application layer requests. This class is what is responsible for progressing the simulation, updating each node, and logging metrics/errors.

NetworkSimulation: Arcade Library

The class extends the `arcade.window` class, which allows it generate a window displaying the simulation world. The class uses the arcade library to draw nodes, links, and text in the window. The arcade library is also used to handle user inputs and movement in the world. The possible user inputs are displayed in the top right of the simulation window

NetworkSimulation: ECR Simulation

The `NetworkSimulation` class is responsible for simulating the network level aspects of the ECR protocol. It handles all the `NetworkNode` objects in the network and progresses them as necessary. The class simulates the links between nodes by providing the functionality to send link maintenance messages. The class also keeps track of all in-flight packets and has each intermediate node handle packets as they are routed to the destination. The class keeps track of the packets that need to be sent during the simulation and tires to schedule them as routes are found. Finally, the `NetworkSimulator` class performs logging of data and performance metrics.