Remy source code Documentation

The classes are defined separately and the functions/structures are defined with the definition

NetConfig:

This class defines the following input parameters as *double* datatype:-
mean_on_duration
mean_off_duration
num_senders
link_ppt
delay
buffer_size
stochastic_loss_rate

Packet:

Keeps the following variables:
-flow_id
-sequence number
-src address
-bookkeeping of ticks

Memory:

- structure for storing input network configurations as follows
    _rec_send_ewma; inter-arrival time of timestamps echoed in ACKs
    _rec_rec_ewma; inter-arrival time of ACKs
    _rtt_ratio; ratio of lowest rtt and last rtt
    _slow_rec_rec_ewma; inter-arrival time of ACKs with a much higher weight for new samples
    _rtt_diff:difference between lowest and latest rtt
    _queueing_delay; product of outstanding pkts and ACKs inter-arrival time

MemoryRange:

Memory _lower, _upper; -stores upper and lower values of memory.

vector< Axis > _active_axis; - saves which input parameters are in use for example a particular sender can use the set {_rec_send_ewma, rtt_ratio} and ignore others.

mutable unsigned int _count; keeps track of the generation of an action

hash_value() → finds the hash(using boost lib) from the upper and lower part of the memory_range

bisect() → function which creates new actions by dividing the memory_range into two(at midpoint) for every input parameter(axis) being in use for that memory_range

Action:

Generic class for an action to be performed for a sender/flow.
unsigned int _generation; denotes how many times it has been used for a flow/sender
MemoryRange _domain; ranges of input parameters that correspond to this action

bisect() → function which creates new actions by dividing the memory_range into two(at midpoint) for every input parameter being in use for that action

Note- There are two types of actions, one which adjusts only the sending rate and other adjusts the 3 variables; window multiple, window increment and min. sending rate

Whisker:

This stores the 3 action variables that remy adjusts for each sender i.e.
1.window increment
2. window multiples
3. minimum sender rate

next_generation() → function to geometrically increment/decrement the values of the actions and returns all the resulting array of whiskers

hash_value(): returns the hash of all the action combined and the memory_range object associated with it.

Struct Optimization Settings:

```
T min_value; /* the smallest the value can be */
T max_value; /* the biggest */

T min_change; /* the smallest change to the value in an optimization exploration step */
T max_change; /* the biggest change */

T multiplier; /* explore multiples of the min_change until we hit the max_change */
/* the multiplier defines which multiple (e.g. 1, 2, 4, 8... or 1, 3, 9, 27... ) */

T default_value; // hardcoded value
```

Note: used in defining Whisker(remy action taken) parameters as well as Fin(sender rate)

WhiskerTree:

Keeps a leaf which are a set of whiskers and a vector of children WhiskerTrees

WhiskerTree(const Whisker & whisker, const bool bisect) → creating new whiskers from supplied one if bisect is true otherwise just add this whisker to leaf. New whiskers have different memory ranges determined by bisect function mentioned before.

most_used(const unsigned int max_generation) → returns the most used Whisker in the WhiskerTree i.e. the most used action corresponding to the most encountered input params

WhiskerTree::reset_counts(void) → sets the count of the number of times a network inputs have been encountered and corresponding action used to 0

reset_generation(void) → reset the generation of all whiskers(or actions) in this object to 0

total_whisker_queries(void) → returns the total number of times that the whiskers(or actions) in this tree were called.

str(const unsigned int total) → return the parameters of all whiskers in string form detailing their action values

WhiskerTree(const RemyBuffers::WhiskerTree & dna) → instantiate a new WhiskerTree using the provided parameter, dna.

Fin:

This is a specialized action class which stores the sender rate, *lambda*.

get_optimizer(void) → defines and returns a hardcoded default value for *lambda*

struct OptimizationSettings → defines the *lambda* as an optimizationsetting struct and provides a function, DNA(void), to return a serialized version of *lambda*

next_generation(void) → provides a list of fins which define variations of the current fin by performing geometric increments/decrements on the current *lambda* value

str(const unsigned int total) → returns the usage, generation of the action and *lambda* value as a string after scaling it on total.

round(void) → rounding *lambda* value to four decimal places

hash_value(const Fin & fin) → hash value using boost lib of lambda with the memory_range object associated with this fin action

FinTree:

Same as WhiskerTree except instead of Whisker objects, we have Fin objects.

Rat:

Defines a flow instance and the associated parameters of a flow such as latest ack number, window size, sending rate, number of pkts sent/received. It also keeps whiskers corresponding to the network connection inputs.

packets_received(const vector< Packet > & packets) → receives a list of packets(acting as network receiver) and updates all the input parameters. It then searches the whisker list to find the corresponding action associated with it and updates the sender window and inter-packets send time.

next_event_time(const double & tickno) → returns the next time packets will be sent

send(const unsigned int id, NextHop & next, const double & tickno,
const unsigned int packets_sent_cap) → this function creates a packet from the passed tick_no and id.

Evaluator:

This is a class responsible for simulating a network of senders

score(): This function simulates a network of senders and returns a score for a given set of actions for a group of senders.

Network:

Contains two groups of senders (which can be time-switched or byte-switched)
Tick(): simulates a group of senders which send data over the network. The packets are dropped with a Bernoulli distribution and a delay is added everytime as it traverses from sender to receiver.

Remy.cc

Simulates a network of senders with a single action. Score is evaluated for that action set and if there is a better action set by incrementing/decrementing geometrically, it is added to the action set tree. At the end, the old action set tree's score is compared to the new one by performing 10 simulations on both of them. The better one is chosen and the score along with input configurations is outputted to a file with a pre-written format.