-

-

Gilbert Loss Trace Generator

Martín Varela - VTT

January 17, 2012

Abstract

This program allows the generation of several accurate (with respect to pre–defined target values) loss traces following a simplified Gilbert loss model (2-states, one with no losses, and one with loss probability = 1). For subjective testing, the challenge lies in getting the right stats within a few hundred packets, as test sequences are only $\sim 10s$ long. We take a brute-force approach, generating several samples per combination of loss–rate and mean loss burst size until enough sufficiently good traces are generated.

1 Preliminaries

We will define our *Main* module and import some standard library functions and types.

```
module Main where
import Data.List
import System.Random
import System.Environment
```

2 Packet Sequences

A network flow is represented by a sequence of packets, which either arrive at their destination, or don't. We model this by the type *Packet*, defined as follows:

```
data Packet = P_OK | P_LOST deriving Eq
instance Show Packet where
  show P_OK = "0"
  show P_LOST = "1"
```

3 Valid Sequences

For a given sequence of packets σ, and target values for the loss rate and mean loss burst size, LR_t and $MLBS_t$ respectively, we consider σ to

be suitable if the difference of the observed loss rate and MLBS in the sequence, LR_σ and $MLBS_\sigma$ and the target values is lower than a given threshold.

We will define the thresholds at a 5% of LR_t for the loss rate, and 0.1 packets for the mean loss burst size., and so our conditions for accepting a sequence as valid are

$$\mid LR_t - LR_\sigma \mid \leqslant 0.05 \times LR_t$$

and

$$\mid MLBS_t - MLBS_\sigma \mid \leqslant 0.1$$

There is an issue when the definition of the mean loss burst size when no losses occur. For some applications, it may be convenient to define it as 1, while in other cases it might be better to define it as 0 (as indeed, if there are no losses, speaking of the mean loss burst size does not make sense). However, for QoE estimation purposes using PSQA or a similar technique, defining it as 0 might be problematic as it introduces a discontinuity in the mean loss burst size axis (i.e. when there are losses, for any loss rate, the mean size of each burst is at least 1). We will, therefore, define the mean loss burst size of a lossless sequence as 1, define our functions accordingly.

We then have:

```
checkSequence :: Double → Double → [Packet] → Bool
checkSequence tlr tmlbs s
  | tlr > 0 = (abs $ tlr − lr s) ⩽ 0.05 ∗ tlr ∧
              (abs $ tmlbs − mlbs s) ⩽ 0.1
  | otherwise = lr s ≡ 0
```

We calculate the loss rate in the sequence by counting the number of lost packets and dividing over the sequence length.

```
lr xs = (fromIntegral ∘ length ∘ filter (≡ P_LOST) $ xs) /
  (fromIntegral ∘ length $ xs)
```

The mean loss burst size is calculated as the average of the lengths of loss events (i.e. instances where one or more packets are lost). To this end we extract the loss events from the sequence, and calculate the average of their lengths. As discussed above, if there are no losses, then we define the mean loss burst size as one.

```
mlbs xs
  | length l_events > 0 = (fromIntegral ∘ sum ∘ map length $ l_events) /
    (fromIntegral ∘ length $ l_events)
  | otherwise = 1
  where
    l_events = (filter (λe → head e ≡ P_LOST)) ∘
      (groupBy (λx y → x ≡ y ∧ y ≡ P_LOST)) $ xs
```

4 Sequence Generation

In order to generate sequences with the desired loss process, we need to calculate, from the target parameters LR_t and $MLBS_t$, the probabilities for the simplified Gilbert model. The conversion is given by

$$p = \frac{1}{MLBS_t} \frac{LR_t}{1 - LR_t}$$

and

$$q = \frac{1}{MLBS_t}$$

where p and q correspond to the probabilities of going from the no–loss state to the loss state, and vice–versa, respectively, as seen in Figure 1.

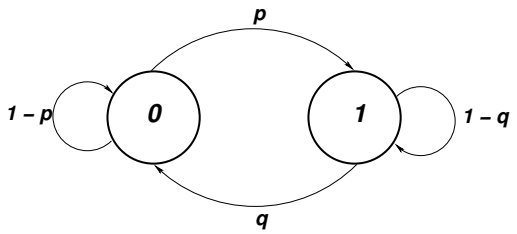

Figure 1: The simplified Gilbert model.

Sequences need to be generated with a pre–defined length, and we will need to obtain several different sequences with similar statistical loss behavior. So if we want to obtain k sequences with a certain loss rate *tlr* and mean loss burst size *tmlbs* within a certain tolerance as defined in Section 3, then we can imagine generating an infinite list of sequences ls with the target parameters and selecting from those the first k sequences that are valid.

We can then write

```
selectSequences :: Int → Double → Double → [[Packet]] → [[Packet]]
selectSequences k tlr tmlbs s = take k $
  filter (checkSequence tlr tmlbs) s
```

It now remains the task of generating the actual sequences with the desired targets. Since it would be useful to obtain repeatable traces, we start by taking a seed as an argument. We'll use that seed to generate a pseudo–random sequence of integer seeds for creating new generators for the actual packet loss sequences. In this way, we get the repeatability, and we keep a larger portion of the code pure.

```
seeds s = (randoms $ mkStdGen s) :: [Int]
```

In order to generate the sequences, we need to implement the two–state Markov chain depicted in Figure 1. We use one of the previously generated seeds to feed a new generator, and use this to simulate the

chain. So, the creation of a sequences takes as arguments the transition probabilities for the Markov chain, the desired sequence length, and a seed for the pseudo–random number generator. We always start from a loss–free state.

```
createSequence :: Double → Double → Int → Int → [Packet]
createSequence tlr tmlbs k s = unfoldr fgen (p, q, P_OK, probs)
  where
    probs = take k $ (randoms $ mkStdGen s) :: [Double]
    p = (tlr / (1 − tlr)) / mbs
    q = 1 / mbs
    mbs
      | tmlbs > 0 = tmlbs
      | otherwise = 1


fgen :: (Double, Double, Packet, [Double]) →
  Maybe (Packet, (Double, Double, Packet, [Double]))
fgen (_, _, _, []) = Nothing
fgen (p, q, current, probs) = Just (next, (p, q, next, tail probs))
  where
    next = case current of
      P_OK → if (p ⩽ head probs)
               then P_OK
               else P_LOST
      P_LOST → if (q ⩽ head probs)
               then P_LOST
               else P_OK
```

Having the means to generate sequences with the desired target loss characteristics, we just create an infinite list of such sequences, from which we will then choose as many as we need. It should be noted that depending on the target values and tolerances, this might result in a non–halting computation, as some combinations of target values and sequence length are not feasible.

```
sequences :: Double → Double → Int → Int → [[Packet]]
sequences tlr tmlbs k s = map (createSequence tlr tmlbs k) $ seeds s
```

With the sequence generation solved, we can now build the rest of the program, which will take arguments for the target loss rate, the target mean loss burst size, the length of the sequences to be generated, the number of sequences to be generated, and a seed for the RNG. The program will then create a file per sequence generated, and a file with the actual loss rates and mean loss burst sizes of the sequences generated, for validation purposes.

```
main = do
  args ← getArgs
  let tlr   = read $ args !! 0 :: Double
    tmlbs = read $ args !! 1 :: Double
    lenS  = read $ args !! 2 :: Int
```

```
    numS = read $ args !! 3 :: Int
    seed = read $ args !! 4 :: Int
  mapM_ (createFile tlr tmlbs seed) $
    zip [1..] (selectSequences numS tlr tmlbs $
      sequences tlr tmlbs lenS seed)
```

The creation of the trace and statistics files is handled like so:

```
createFile :: Double -> Double -> Int -> (Int, [Packet]) -> IO ()
createFile tlr tmlbs seed (seqno, s) = do
  let outfile = concat ["trace_"
                       , show tlr
                       , "_"
                       , show tmlbs
                       , "_"
                       , show seed
                       , "_"
                       , show seqno
                       , ".txt"]
      statsfile = concat ["stats_"
                       , show tlr
                       , "_"
                       , show tmlbs
                       , "_"
                       , show seed
                         , ".txt"]
      stats     = concat ["Sequence "
                       , show seqno
                       , " lr = "
                       , show $ lr s
                       , ", mlbs = "
                       , show $ mlbs s
                       , "\n"]
  writeFile outfile $ concat $ map (show) s
  appendFile statsfile stats
```