API Documentation

### 1 Introduction to the API

This API serves the purpose of providing the users a simple and intuitive way to access the information contained in a dataset. This user-centric approach aims to abstract the client as much as possible from the internal details the dataset, thus making the process of gathering information convenient, and training-free. The main idea is that the tool provides structured samples in a sequential manner, and it is up to the user to extract the desired information for each sample and use it.  
  
In the following sections, a detailed description of the Sample instances will be given. The idea is that we have a container to store the information regarding the topology of the network, the parameters of the traffic distributions used to generate the traffic, the routing implemented by the nodes, and the resulting delay, jitter and drops simulated from the configured network. All this information is provided in a source-destination fashion.

### 2 How does it work?

The user, in its code, needs to initialize an instance of the Parsing Tool as follows:  
  
*tool* = *ParsingTool*(*< pathToDataset >,< desiredIntensityRange >*)  
  
Where *pathToDataset* states where the uncompressed dataset file resides, and *desiredIntensityRange* is an array of integers, where the user can specify a single value, if only a single intensity is desired, or two values, that will be the lower and upper bounds for the intensities desired. If desiredIntensityRange is an empty array, all intensities will be considered as desired.  
  
Afterwards, the following line will return an iterator object:  
  
*it* = *iter*(*tool*)  
  
Once the user has this iterator object, there are two main ways to use it:

* *next*(*it*): Provides the next sample object of the iterator.
* *for out in it*: Loop structure that will return sample objects until no more information can be processed.

As expected, the first method requires to manually ask for a sample each time it is needed, but allows for a greater granularity, since each sample can be treated differently. The second method enforces a similar treatment for each sample and is a more convenient way if the objective is to process all the dataset.

### 3 Structure of a Sample

The final product of the API is an iterator, that allows the user to process the dataset, and to obtain a Sample instance, for each combination of topology, routing, and traffic matrice. The instance will be comprised of the following attributes:

* *pkt*\_*by*\_*network*: Packets transmitted in the network per time unit (second).
* *loses*\_*by*\_*network*: Packets lost in the network per time unit (second).
* *delay*\_*network*: Average delay of all paths in the network.
* *result*\_*matrix*: Matrix with aggregated and flow-level information about losses, delay and jitter for each pair [*src,dst*].
* *traffic*\_*matrix*: Matrix with time and size distribution of the generated traffic for each pair [*src,dst*].
* *routing*\_*matrix*: Matrix with the path to go from node *i* to node *j* for each pair [*src,dst*].
* *topology*\_*matrix*: Matrix with the bandwidth for each pair [*src,dst*] if both nodes are adjacent, or -1 otherwise.

For the case of *result*\_*matrix*, further explanation has to be done concerning its structure. Its architecture is the following:

* *AggInfo*: dictionary with aggregated information for all flows between the specific [*src,dst*].
  + *PktsDrop*: packets dropped per time unit over the path [src,dst].
  + *AvgDelay*: average per-packet delay over the packets transmitted over the path [src,dst]
  + *AvgLnDelay*: average per-packet neperian logarithm of the delay over the packets transmitted in the path
  + *p*10*,p*20*,p*50*,p*80*,p*90: percentiles of the per-packet delay over the path
  + *Jitter*: Variance of the per-packet delay (jitter) over the packets transmitted over the path
* *Flows*: List of dictionaries with flow-level information for each flow between node *i* and node *j*.
  + *PktsDrop*: packets dropped per time unit of the flow.
  + *AvgDelay*: average per-packet of the flow delay
  + *AvgLnDelay*: average per-packet of the flow neperian logarithm of the delay
  + *p*10*,p*20*,p*50*,p*80*,p*90: percentiles of the packets of the flow
  + *Jitter*: variance of the per-packet of the flow delay (jitter)

Then, assuming *r* is the Result matrix of an arbitrary iteration, we could access it as follows:

* *r*[*src,dst*]: dictionary with the information for the communication between node *src and* node *dst*.
* *r*[*src,dst*][′*AggInfo*′]: dictionary with the aggregated information for the communication between node *src* i node *dst*. Extend to *r*[*src,dst*][′*AggInfo*′][′*nameparam*′], substitute the last field with any of the keys stated previously, to obtain the specific value for that parameter.
* *r*[*src,dst*][′*Flows*′]: flow-level information for the communication between node *src* i node *dst*. Extend to *r*[*src,dst*][′*AggInfo*′][′*numflow*′]′ and substitute the last field with the number of the desired flow to obtain its information. Extend to *r*[*src,dst*][′*AggInfo*′][′*numflow*′][′*nameparam*′] and substitute the last field with any of the keys stated previously, to obtain the specific value for that parameter, for that specific flow.

As for *traffic*\_*matrix*, more explanation is also needed. It is structured as follows:

* *AggInfo*: dictionary with the following aggregated information of all the flows between the specific [*src,dst*]:
  + *AvgBw*: average banndwidth in Kbps of all the traffic from src node to dst node.
  + *PktsGen*: packets generated from src node to dst node per time unit.
* *Flows*: List of dictionaries with flow-level size and time distribution information for each flow between node *i* and node *j* (See relation between list of parameters for each time and size distribution below).
  + *TimeDis*: time distribution used by the flow.
  + *TimeDistParams*: dictionary with the parameters of the specific time distribution.
    - *EqLambda*: average bits per second to generate.
    - *AvgPktsLambda: average number of packets of average size generated per time unit.*
    - *MinPktLambda*: minimum number of packets of average size generated per time unit.
    - *MaxPktLambda*: maximum number of packets of average size generated per time unit.
    - *StdDev*: Standard deviation of the normal of the interarrival time.
    - *PktsLambdaOn*: Average packets per second to generate during the ON period.
    - *AvgTOff*: Average time OFF of the burst. Uses exponential distribution.
    - *AvgTOn*: Average time ON of the burst. Uses exponential distribution.
    - *ExpMaxFactor*: Factor used to define the maximum valid value of the exponential, defined as *factor* \* *average*\_*time*.
    - *BurstGenLambda*: Lambda rate of burst generation.
    - *Bitrate*: Rate of the burst flow: bps.
    - *ParetoMinSize*: Minimum number of bits in the burst.
    - *ParetoMaxSize*: Maximum number of bits in the burst.
    - *ParetoAlfa*: Tail index.
  + *SizeDist*: size distribution used by the flow.
  + *SizeDistParams*: dictionary with the parameters of the specific size distribution.
    - *AvgPktSize*: Average size of the packets in bits.
    - *MinSize*: Minimum size of the packets in bits.
    - *MaxSize*: Maximum size of the packets in bits.
    - *PktSize*1: First possibility of packet size.
    - *PktSize*2: Second possibility of packet size.
    - *NumCandidates*: Number of different packets size considered.
    - *Size*\_*i*: Size of the candidate packet *i*.
    - *Prob*\_*i*: Probability to use candidate packet *i*.
  + *AvgBw*: average bandwidth for this flow.
  + *PktsGen*: packets generated by this flow per time unit.

*traffic*\_*matrix* accessing is done in the same fashion as in the Result Matrix, but this time each flow will contain a dictionary with the parameters described above. Refer to the list of parameters to correlate every size and time distribution with its associated values.  
  
The *routing*\_*matrix* accessing process is pretty much straightforward. Assuming *rout* is a *routing matrix*, using *rout*[*src,dst]* will return a string describing the path to go from node *src* to node *dst*.  
  
The *topology*\_*matrix* accessing process is equivalent to the previous one. Assuming *topo* is a *topology*\_*matrix*, *topo*[*src,dst*] will return the bandwidth of the link that connects node *src* and node *dst*, if they are directly adjacent, or -1 if they are not.

#### 3.1 Parameters of time distributions

* *Exponential*: *EqLambda*, *AvgPktsLambda*, Exp*MaxFactor*
* *Deterministic*: *EqLambda*, *AvgPktsLambda*
* *Uniform*: *EqLambda*, *MinPktLambda*, *MaxPktLambda*
* *Normal*: *EqLambda*, Avg*PktsLambda*, *StdDev*
* *OnOff*: *EqLambda*, *PktsLambdaOn*, *AvgTOff*, *AvgTOn*
* *PPBP*: *EqLambda*, *BurstGenLambda*, *Bitrate*, *ParetoMinSize*, *ParetoMazSize*, *ParetoAlfa*, *ExpMaxFactor*

#### 3.2 Parematers of size distributions

* *Deterministic*: *AvgPktSize*
* *Uniform*: *AvgPktSize*, *MinSize*, *MaxSize*
* *Binomial*: *AvgPktSize*, *PktSize*1, *PktSize*2
* *Generic*: *AvgPktSize*, *NumCandidates*, *Size*\_*i*, *Prob*\_*i*

#### 3.3 Methods offered by Sample instances

The sample object offers methods to obtain the matrices resulting of each iteration of the simulation, and the dictionaries for each one of the source-destination pairs, for each one of the matrices. Hence, the user can use the following methods (assuming that *s* is the sample obtained for a single iteration):

* *s.get*\_*result*\_*matrix*(): Returns a matrix *m* such that *m*[*src,dst*] contains a the information that the Result matrix stores for the requested [*src,dst*] pair regarding delay and jitter.
* *s.get*\_*srcdst*\_*result*(*src,dst*): Directly returns the delay and jitter information that the Result matrix stores for the requested [*src,dst*] pair structured as a dictionary. See contain of the dictionary in the previous section.
* *s.get*\_*traffic*\_*matrix*(): Returns a matrix *m* such that *m*[*src,dst*] contains a dictionary with the information that the Traffic matrix stores for the requested [*src,dst*] pair regarding the traffic generated from the src node to dst node.
* *s.get*\_*srcdst*\_*traffic*(*src,dst*): Directly returns a dictionary with information that the Traffic matrix stores for the requested [*src,dst*] pair. See contain of the dictionary in the previous section.
* *s.get*\_*routing*\_*matrix*(): Returns a matrix *m* such that *m*[*src,dst*] returns the information that the Routing matrix stores for the requested [*src,dst*] pair regarding the path that links both nodes.
* *s.get*\_*srcdst*\_*routing*(*src,dst*): Returns a string with the routing path between the src and dst node.
* *s.get*\_*topology*\_*matrix*(): Returns a matrix *m* such that *m*[*src,dst*] returns the information that the Topology matrix stores for the requested [*src,dst*] pair, that is, the bandwidth between both nodes if they are adjacent, or -1 otherwise.
* *s.get*\_*srcdst*\_*topology*(*src,dst*): Returns the bandwidth between both nodes if they are adjacent, or -1 otherwise.