Get Started with the ICN Evaluator (ICE)

Mays AL-Naday

2018-03-14

Overview

ICE package provides a framework to model and analyse Pub/Sub based Information-Centric Networks (ICN), and IP/HTTP-over-ICN networks. The package allows for modeling multiple service/content/information publishers (i.e. service points); offering to multiple subscribers (consumption points). Service provision in ICN or IP/HTTP-over-ICN can either be unicast or multicast. The package also allows for modelling service provision over tranditional IP networks, such as in current Content Distribution Networks (CDNs). The network scenario is provided through a YAML-based configuration file in `tools/`

Installation

To install ICE:

- make sure to have R 3.4.0+ and that `gfortran` is installed
- open R session
- install ICE from github:
  - clone the source pacakge from GitHub repo `point-evaluation-tools`, using branch `ice`:

```
git clone git@github.com:point-h2020/point-evaluation-tools.git . -b ice
```

  - open an R session and install :
    - devtools (if not installed alread):

```
install.packages('devtools')
```

    - ice:

```
devtools::install()
```

Configuration

Scenarios attributes are provided in a YAML-based configuration file. The file is devided into `enviroments`, those are sections that define attributes for different purposes. enviorments in current ICE version include:

- `default`: the default (parent) enviroment, which provides default values for all attributes. It is also used to provide configs for generating input data (as will described in `Simulating with ICE`)
- `simulate`: defines any special values for simulating the scenario, that is provisiong services from publishers to subscribers.

default

The `default` enviroment describes the scenario attributes:

- `cores`: number of CPU cores to be used in parallel during data generation and simulation
- `dirs`: paths to store generated input data, and simulation results. Those are:
  - `home`: path to home directory for all data
  - `store`: path to store input data
  - `result` path to store simulation results

example of the top-level section in **config.yml**:

```
default:
  cores: 2
  dirs:
    home: "./"
    store: "data/"
    result: "data/results/"
```

- `graph`: this section specifies the network topology information (i.e. graph). Current ICE v1.0.0 only supports graphs imported from the Interent Topology Zoo dataset. Graph files should be in either `graphml` or `gml` format.
  - `gdir`: path to where graphs are stored
  - `gset`: set of indecies that point to specific graphs in the directory, e.g. `gset: [1]` refers to graph Abilene in the `zoo` list of graphs
  - `augument`: whether to augument the network by attaching access networks. The option is not used in current ICE v1.0.0

  example of `graph` section in **config.yml**:

```
default:
  graph:
    gdir: "data-raw/zoo"
    gset: [1]
    augument: NULL
```

- `catalogue`: specifies the service/content/information catalogue attributes, including:
  - `size`: number of items in the catalogue
  - `v`: storage volume and/or bitrate of item. A set of values is provided to randomly choose from (per item). The storage unit is GB, while the bitrate unit is Gbps
  - `d`: popularity disribution. Current ICE v1.0.0 only supports `zipf`
  - `d.param`: distribution parameters. Notice, this subsection changes depending on the distribution of `d`, for Zipf it includes:
    - `s`: Zipf exponent. Providing multiple exponents allows for simulating the same catalouge but with different exponents. Usefull for condusting senstivity analysis
    - `ig`: ignored ratio, which corresponds to a fraction of the distribution to be omitted before the disribution is used to generate the items' popularity

  example of `catalogue` section in **config.yml**:

```
default:
  catalogue:
    size: [1000]
    v: [0.02,0.04, 0.06]
    d: "zipf"
    d.param:
      s: [0.85]
      ig: 0.0
```

- `origin`: specifies attributes of origin publishers, those are large data centres with cached copy of every content/service/information. Origins can also represent Cloud datacentres with negligible constraints on storage/processing/networking resources. Attributes include:
  - `opt`: selection policy (algorithm). ICE v1.0.0 provides three policies:
    - `Pop` psudo random selection with higher priority to nodes of largest populations,
    - `Cls` psudo random selection with higher priority to nodes of highest connectivity (closeness), and
    - `Swing` determinstic selection swining between nodes of higher and lower connectivity
  - `k`: number of origins in the network - i.e. number of nodes to be selected.
  - `a`: minimum storage capacity that can be placed in a node - i.e. storage atom. E.g. `a: 0.1` means only multiples of 0.1 GB can be placed in the node

example of `origins` section in **config.yml**:

```
default:
  origin:
  opt: [1]
  k: [2]
  a: 0.1
```

- `surrogate`: similar to `origin`, the section specifies attributes of surrogate publishers, those are constrained service points at the edge (closer to end-users) with cached copy of only a subset of items from the catalogue. Surrogates can also represent Fog nodes with high constraints on storage/processing/networking resources. Attributes include:
  - `opt`: selection policy (algorithm). ICE v1.0.0 provides three policies:
    - `Pop` psudo random selection with higher priority to nodes of largest populations,
    - `Cls` psudo random selection with higher priority to nodes of highest connectivity (closeness), and
    - `Swing` determinstic selection swining between nodes of higher and lower connectivity Specifying multiple algorithms allows for simulating for different algorithms. This is usefull to compare behavious with different algorithms
  - `k`: number of surrogates in the network - i.e. number of nodes to be selected.
  - `a`: minimum storage capacity that can be placed in a node - i.e. storage atom. E.g. `a: 0.1` means only multiples of 0.1 GB can be placed in the node

  example of `surrogate` section in **config.yml**:

```
default:
  surrogate:
  opt: [1]
  k: [2,4]
  a: 0.1
```

- `dns`: specifies attributes of DNS points in traditional DNS-based networks. Attributes include:
  - `opt`: selection policy (algorithm). ICE v1.0.0 provides three policies:
    - `Pop` psudo random selection with higher priority to nodes of largest populations,
    - `Cls` psudo random selection with higher priority to nodes of highest connectivity (closeness), and
    - `Swing` determinstic selection swining between nodes of higher and lower connectivity Specifying multiple algorithms allows for simulating for different algorithms. This is usefull to compare behavious with different algorithms
  - `k`: number of DNSs in the network - i.e. number of nodes to be selected.

  example of `dns` section in **config.yml**:

```
default:
  dns:
  opt: [1]
  k: [2,4]
```

- `permutation`: specifies testing conditions in the network, including:
  - `tests`: number of permutations (random generations) to generate
  - `load`: the network load, described as a fraction of the total number of users in the network. the parameter is provided with 3-elements vector, giving:
    1. the lower bound on the network load
    2. the upper bound on network load
    3. increment value from lb to up
  - `v`: specifiy which network technology to simulate, choises are:
    - `ip`: for traditional IP and DNS networks
    - `icn.uc`: ICN network with unicast delivery
    - `icn.mc`: ICN network with multicast delivery

  example of `permutation` section in **config.yml**:

```
default:
  permutation:
    tests: 50
    load: [0.4,0.4,0.1]
    v: []
```

simulate

The `simulate` enviroment is a child enviorment of `default` and inherts all attributes that are not re-specified here. This enviroment is used to specify the simulation parameters, a subset of the generation parameters specified in `default`. The reason for having a separate enviroment for simulation is to have the flexibility to run simulations for a subset of the generated permutations. The enviorment also specifies simulation attributes that are of no relevance during data generation, those extend the `permutation` section, include:

- `sim.tests`: number of permutations to simulate. This *must be* `<= tests`
- `L`: streaming total period, this is the time require to transfer all the chunks of data from the service to the consumption point
- `t`: catchement interval, that is the period to group users within in one multicast group

Example of `permutation` section in `simulate` enviroment of **config.yml**:

```
default:
  permutation:
    sim.tests: 10
    L: [900, 1800]
    t: [0.1, 1.0]
    v: ['ip', 'icn.uc', 'icn.mc']
```

Workflow

ICE workflow is split into four distinct phases. To enable each phase, there is a set of wraper functions that allows for achieving multiple tasks, by calling other functions within. Here, I use those functions with example inputs to the arguments and describe of some of the 'not-so-clear' arguments. Notice two things:

- ICE provides logging capabilities via the `logging` library. Each function that may have own logging level would be passed the argument `config.active`
- Values assigned to the arguments in below functions are **NOT** default values, they are just illustrative example input

Parsing

First thing ICE would expect is the attributes of the network (and scenario) to be simulated. Remeber, those are descirbed in config.yml. ICE provides a wrapper function `ParseConfig` that creates configs enviroments providing all elements expeceted by ICE.

```
gcfgs <- ParseConfig(config.file = 'tools/config.yml', config.active = 'INFO', config.env = 'default')
scfgs <- ParseConfig(config.file = 'tools/config.yml', config.active = 'INFO', config.env = 'simulate')
```

Notice here that it is possible to have two different configurations enviroments, one for data generation and one for simulation. This brings us back to the flexibility mention above in allowing for simulation to happen over a subset of the generated data. ### Generation This phase generates and store input data, most important of which is the Pub/Sub State per item, which will be used in modelling the network Data. Input data will be stored in the path specified by `store` argument in configuration, under a predefined file structure, described in Design Description vignette.

Once ICE has a configuration enviroment `cfgs`, next would be to generate the input data and simulate the scenario. in ICE v1.0.0 data generation and simulation are distinct separate phases. This distinction is clearly reflected in the design of the wrapper functions below. To have a different workflow - e.g. to mix generation and simulation at a finer level - one would just need to create new wrapper function(s) that calls the inner functions in a different way.

The wrapper function used for generating input data:

```
GenSimData(cfgs = gcfgs)
```

This function will check if input data is stored and if not, then generate and store them.

Simulation

That is when the Pub/Sub states will be routed in the network, leading to provision storage, processing and/or networking resoruces. Once data is guaranteed to be stored as expected, simulations can be run. Notice that paths and filenames of input data is described Design Description vignette.

Simulation can be specified according to the network techonology, as described in the Configuration section. For each technology, there is a wrapper function:

- IP/DNS-based network:

  ```
  SimIp(cfgs = scfgs)
  ```

- ICN or IP-over-ICN based network with unicast transmission:

  ```
  SimIcnUnicast(cfgs = scfgs)
  ```

- ICN or IP-over-ICN based network with multicast transmission:

  ```
  SimIcnMulticast(cfgs = scfgs)
  ```

Simulation results will be saved in the path specified by `result` argument in **config.yml**, sub-directories and filenames is described in Design Description vignette.

Processing

After simulation, raw results of reserved resources and provisioned paths are then processed to generate 'knolwedge' of the network performance. ICE v1.0.0 comes with a set of functions to generate knowledge about reserved network capacity and ECDF of paths' length. Notice here that the configuration enivroment used for simulation is the one that **must** be used - in our example that is `scfgs` - not the configuration used for generation. That is because, one cannot process for attributes that have not been simulated yet!

Network Capacity

- IP/DNS-based mean reserved capacity for unicast delivery:

  ```
  ProcIpCapacity(cfgs = scfgs)
  ```

- ICN mean reserved capacity for unicast delivery:

  ```
  ProcIcnUnicastCapacity(cfgs = scfgs)
  ```

- ICN mean reserved capacity for multicast delivery:

  ```
  ProcIcnMulticastCapacity(cfgs = scfgs)
  ```

Path Length

- IP/DNS-based path length (ECDF):

```
ProcIpPathLength(cfgs = scfgs)
```

- ICN unicast path length (ECDF):

```
ProcIcnUnicastPathLength(cfgs = scfgs)
```

Total Storage

Total storage requirements for all items, which is then split between that is provided for caching and that which is not.

```
ProcTotalStorage(cfgs = scfgs)
```

Summary

The above gives a start point for how to use ICE for modelling *-over-ICN and IP/DNS-based networks. To understand more details ICE internals, check `Design Description` vigennet. To go through a detailed example of simulating a network performance, check the `Examples` vigenette.