University College London
Department of Computer Science
COMP0023: Networked Systems

Individual Coursework 2: Routing

Due: 12th December 2025

The Internet is composed by many independent networks, called Autonomous Systems (ASes) or domains. To ensure that packets can be delivered from any host in the Internet to any other, network devices in all ASes run protocols that enable control-plane message exchanges and population of their respective forwarding tables. Switches compute paths at the link layer, within LANs; routers enable packet forwarding within each domain, running so-called Interior Gateway Protocols (IGPs), and across multiple domains, running the Border Gateway Protocol (BGP). BGP, in particular, is the de-facto standard protocol for inter-domain routing. It enables the computation of paths across competing ASes, implementing collaborative routing under competitive pressure.

Overview

In this coursework, we will focus on the design and implementation of a simplified and partially modified version of BGP, which we refer to as EGP (for Exterior Gateway Protocol). Such a version models the basics of inter-domain routing, ignoring many advanced features of actual BGP implementations. It also relies on minimalistic message syntax and router configurations.

Because we won't be able to offer direct access to routers whose software can be arbitrarily modified, we'll do the next-best thing: we'll use virtual routers and run network simulations.

We assume that you have access to a laptop, a desktop or a UCL machine with decent Command Line Interface (CLI), such as bash, zsh or an equivalent Windows command prompt.

As a starting point, we provide you with the implementation of a simple discrete-event simulator that can model networks of inter-connected routers.

The simulator requires a basic Python3 installation, including only standard libraries plus the `networkx` Python module. For documentation and installation instructions about `networkx`, please refer to the official documentation at `networkx.org`.

We now provide an overview of our simulator.

Simulator. In each run, the simulator reads a configuration file (specified through the `-c` CLI parameter) that describes the topology of the network to be simulated, as well as the events that will occur during the simulation. It then simulates the specified network by running one copy of your routing code for each virtual router, and passing packets between routers over the links specified in the configuration.

All your code must be included in `egp.py`: this is the only file with code that you will submit!
Your code must rely solely on Python3 standard libraries and possibly `networkx`: do not use any other library or external software.

The interaction between your code and the simulator is fixed, and you must not change it. We now briefly provide an overview of such an interaction; please refer to the code for additional details.

At the beginning of each simulation, a reference to `EGP` objects is provided to already implemented `Router` objects. Each `EGP` object will also be given a reference to a `ForwardingTable` object that models the forwarding table of the corresponding virtual router. During simulations, the `Router` objects will then call specific methods of your `EGP` class – i.e., the ones listed in `AbstractRoutingDaemon` your EGP must be a sub-class of. This interaction will enable your EGP implementation to (i) generate routing messages that will be sent to neighbouring routers, (ii) receive routing messages from neighbouring routers, (iii) know about the status of local links and the revenues obtained by sending traffic over each of them, and (iv) implement all the functionalities you will be asked to implement in this coursework. Your EGP implementation is expected to modify the forwarding table of the virtual router on which it runs, so that user packets can be eventually delivered to their respective destinations, over paths consistent with the exchanged EGP routing messages.

We stress that you can arbitrarily modify any module or class of the simulator while implementing your solution. For example, you can do so to add logging or testing functionalities. However, always remember to keep your solution compatible with the original version of the simulator, as your solution will be marked using the original simulator.

Getting Started

To get started, download and unpack the coursework archive (i.e., tarball) for your coursework. If you are using Unix as OS, you can for example do so from the command line by executing the following commands:

```
$ wget www0.cs.ucl.ac.uk/staff/S.Vissicchio/comp0023/cw2/<portico-student-number>.tar.gz
$ tar xvzf <portico-student-number>.tar.gz
```

where `<portico-student-number>` has to be replaced with your Portico student number as it appears on Moodle. The tarball can also be downloaded by opening the above link on a Web browser.

Unpacking the tarball will create a directory containing all the files needed to complete this coursework. They include: (i) a directory `simulator/`, including the source code for the network simulator; (ii) a skeleton of the EGP class, in the file `egp.py` inside the `lib/` folder inside `simulator/`: you can use such a skeleton as a basis to implement your protocol; (iii) a directory `tests-configs/` including configuration files to test your solution; (iv) a file `baseline-metrics.txt` and a directory `baseline-traces/` with the output of a baseline solution on the tests in `tests-configs/`: your EGP will be evaluated against such a baseline solution;

To run the simulator, you need to have the **networkx** library installed. You can easily install networkx with `pip`. For more detailed information and additional documentation about this library, please refer to *https://networkx.org*. Once the library is installed, you can run the simulator from the CLI as follows.

```
$ python3 simulator.py -c <configuration-file>
```

where `<configuration-file>` is a JSON file describing the simulation to be run, such as any file in `tests-configs/`. You can additionally use options `-v` and `-i` to print more information about the simulation, including revenues calculations and message exchanges.

We believe that the interaction with the simulator and the format of the provided configuration files is quite intuitive, so we skip all the gory and boring details. Feel free however to contact us if you have any issue in running the simulator or questions on the configuration files.

Stage 1: Implement a simplified BGP

Your first task is to mimic BGP behaviour assuming the standard customer-provider-peer policies explained in COMP0023. As a reminder, such policies reflect commercial agreements between ASes, and are revenue-driven. In fact, your EGP has the main goal of maximising revenues: it will have to do so by implementing careful selection and propagation of the routes propagated by its neighbours.

Task. Within the provided network simulator, design and implement a routing daemon that selects and propagates routes so as to maximise revenues, mimicking Internet inter-domain routing.

[60 marks]

- **Specification**. Implement your routing daemon modifying the `EGP` class inside `simulator/lib/egp.py`.

  Your EGP must be able to communicate with EXT objects (see `simulator/lib/ext.py`) which represent routers in neighbouring ASes. This implies processing the messages sent by EXT objects, and sending messages they are able to process. The EGP should then choose the route to use for each destination and update the forwarding table accordingly.

  To do so, you can modify any method specified by the `AbstractRoutingDaemon` class. Among them, the method `processRoutingPacket` is called (by Router objects) for each routing packet destined to your EGP: implement there your logic to process routes sent by neighbouring routers. In addition, the method `generateRoutingPacket` is used to send routing messages to neighbouring routers, and is called once for each interface in every time step: use this second method to announce routes to neighbouring routers.

  Regarding the other methods, `bindToRouter` provides router-specific information, including a reference to the corresponding forwarding table; the `setParameters` and `update` methods offer visibility on the initial parameters set by the simulation configuration, and the (current) state of the interfaces local to the corresponding virtual router.

- **Assumptions and constraints**. Your primary goal is to maximise revenues – as if you were in charge of inter-domain routing within an Internet AS! In each configuration file, revenues are specified per link: they quantify the amount of money you will earn (if the value is positive) or pay (if the value is negative) for each unit of traffic traversing the link, in either direction.

Revenues are computed at every simulation step. For all the details about how revenues are computed, you can check the `EGPChecker` class in `simulator/lib/checkers.py`. We now provide an overview of the most important aspects of such a computation.

For each pair of EXT object and every destination specified by the simulation configuration, revenues associated to every link crossed by the traffic are summed. This sum is multiplied by the amount of traffic sent over each path: such traffic amount is proportional to the number and size of destination prefixes "originated" by the source and destination EXT objects, and is also influenced by the length of the AS path selected by the traffic source – longer AS paths attract less traffic, so they decrease the revenues.

In maximising your revenues, however, you should also pay attention of not breaking basic clauses of (assumed) contractual agreements with your neighbours. They include, but are not restricted to:

– provide full connectivity to customers, for both ingress and egress traffic;

– do not originate routes for prefixes that you do not own;

– do not propagate routes that are not consistent with external messages;

– do not propagate routes that are not consistent with local forwarding entries;

– do not use or propagate routes that would break packet forwarding;

– do not forward traffic over a failed link.

If you try any of the above, your EGP's revenues will decrease, since penalties will be applied. In the case the released code overlooks or has a bug for some obvious checks (e.g., about the AS paths propagated by your EGP), we reserve the right to amend the revenue computation during the marking process. If you are unsure on whether some optimisations performed by your EGP are legit or not, ask us!

Note that it is generally not possible to have forwarding paths *always* consistent with control-plane route, because routing packets are not exchanged instantaneously between routers. In the CW2 simulator, this is captured by the fact that every packet takes one time step to cross a link.

You can assume that your EGP is the only routing protocol that will be used in the network for the considered destinations. As such, your EGP fully controls forwarding in the simulated network: no other portion of code modifies the forwarding table of EGP routers during any simulation.

You can also assume that all the neighbours of your EGP are EXT objects. Note that the EXT class implements a very naive version of the inter-domain routing protocol your EGP should also support. For example, EXT objects assume that they have a single neighbour, and tend to always prefer routes announced by such a neighbour (with the exception of private routes they don't even announce). So, you can take inspiration from EXT objects for your EGP, but you will probably want to extend and modify the logic, eventually. Additionally, your EGP is allowed to take advantage of the naivity of its EXT neighbours – after all, inter-domain routing is a fierce, cinical world!

Additional requirements for your solution follow.

– your solution must rely on **no external library or software** other than standard libraries included in a basic Python3 distribution. The only exception is the **networkx** library, which is already used in the simulator. We will mark your submission in an environment that matches these constraints.

– you must not change the *software interface* of the `EGP` class, nor add functionalities needed by your solution to other classes. We will indeed mark your submission by taking the original simulator in your coursework tarball, and replacing `egp.py` with your submitted file. You can however change each and every line of any class in the simulator to develop, debug and test your solution.

– your solution should enable the simulator to finish each individual test released in the coursework tarball in at most 10 seconds. The same time constraint applies to hidden tests as they model networks of similar size and scenarios of similar complexity as the released tests.

- **Marking scheme**. Your solution will be marked by comparing its behaviour against a baseline solution. Referring to the test labels used in `baseline-metrics.txt`, marks will be assigned as follows:

  – handling customers and providers, including A1-A3 tests [15 marks] and hidden ones [5 marks].

  – interaction with customers, providers and transit-free peers, including B1-B3 tests [24 marks], and hidden ones [16 marks].

We will assign full marks if your solution complies with the above constraints, and performs equal or better than the baseline solution. Performance of the baseline solution for the released tests are specified in `baseline-metrics.txt`. The traces in `baseline-traces/` provide a reference of what is simulated in each test, and of the outcome of the baseline solution at different times.

Partial marks will be awarded depending on the misbehaviours exposed by our tests. As a rule of thumb, you will receive most marks if your solution complies with the above specifications, and exhbits slightly worse performance than the baseline solution. Note that tests are mostly independent from each other: you *don't need* to implement the perfect solution for one test before working on the following ones.

We will use hidden tests to ensure that your code is not an overfit for the released tests. Hidden tests can for example check that your EGP works on different network topologies than the provided ones, that it correctly deals with simulated events similar to those in the released tests, or that it handles a wider set of network settings.

Stage 2: Support advanced peers

The previous stage reflects settings where commercial agreements between ASes and the resulting inter-domain policies closely follow the customer-provider-peer model presented in COMP0023. In the real-world, commercial agreements only depend on the neighbouring ASes: they are basically private contracts, so they can be arbitrarily nuanced and complex.

In this stage, we will therefore explore how to handle a specific new type of commercial agreement that is fictitious but not totally unrealistic. We will do so by introducing "advanced peers".

Intuitively, your EGP will receive positive or negative revenues from any link with an advanced peer, depending on the overall traffic balance with that peer. More precisely, the amount of revenues generated by each advanced peer is specified by the configuration file using the format "X;Y", which indicates that the advanced peer generates X revenues (typically positive) if they send more traffic to the EGP than what they receive, and Y revenues (typically negative) otherwise. This is intended to model a business agreement where the network which generates more traffic pays a fee to the other.

Task. Within the provided network simulator, modify your EGP routing daemon so as to maximise revenues in the presence of advanced peers.

[20 marks]

- **Specification**. Modify the previously implemented `EGP` class inside `simulator/lib/egp.py`. After this second stage, your EGP must be able to interact with any neighbour, including both customers, providers, "standard" peers, and advanced ones. This will not require that you modify message handling, as advanced peers use the same messages as any other neighbouring router. However, you may want to adjust your route selection and propagation to optimise revenues.
- **Assumptions and constraints**. All the assumptions and constraints specified in stage 1 still hold.
- **Marking scheme**. As for stage 1, your solution will be marked by comparing its generated revenues against the revenues generated by our baseline. Marks will be assigned as follows:
  - interaction with customers, providers, transit-free peers and advanced peers, including C1-C4 tests [10 marks], and hidden ones [10 marks].

We will assign full marks if your solution complies with the above constraints, and performs equal or better than the baseline solution. For the released tests, our baseline revenues are specified in `baseline-metrics.txt`. The traces in `baseline-traces/` provide a reference of what is simulated in each test, and of the outcome of the baseline solution at different times.

Partial marks will be awarded depending on the misbehaviours exposed by our tests. As a rule of thumb, you will receive most marks if your solution complies with the above specifications, and exhbits slightly worse performance than the baseline solution. Note that tests are mostly independent from each other: you *don't need* to implement the perfect solution for one test before working on the following ones.

We will use hidden tests to ensure that your code is not an overfit for the released tests. Especially with the addition of advanced peers, the space of possible settings and optimizations becomes quite large. Hidden tests will be used to check the robustness of your solution across such a space, and to reward solutions that better cover a larger set of scenarios.

Stage 3: Predict the baseline's outcome

The previous two stages should help you build a good intuition of the logic and behaviour of the baseline solution. You should thus be able to predict how the baseline solution will behave in settings similar to the ones documented by the released traces, and what is the impact of such behaviour on both routing and forwarding.

Task. Answer the questions included in the file `stage3/stage3-questions.txt`. These questions refer to the behaviour of the baseline solution and its impact on routing and forwarding in the simulation setting specified by `stage3/stage3-config.json`.

[20 marks]

- **Specification**. Create a plain-text file `stage3-answers.txt` reporting your answers to the questions in `stage3/stage3-questions.txt`. The file `stage3-answers.txt` must have the following format:

  ```
  Q1. <max-150-words>
  Q2. <max-150-words>
  Q3. <max-150-words>
  Q4. <max-150-words>
  Q5. <max-150-words>
  ```

  where Q1, Q2, Q3, Q4 and Q5 respectively correspond to each question in `stage3/stage3-questions.txt`, and `<max-150-words>` must be replaced with your answer to the corresponding question. As the above format suggests, each provided answer must consist of *at most* 150 words: shorter answers would be better.

- **Assumptions and constraints**. The file `stage3-answers.txt` must be a plain text file: no rich text format (rtf), Windows doc or other format will be accepted.

  Your answer must be brief (i.e., up to 150 words), concise and to the point. If you end up writing more than 150 words for any answer, you should interpret this as a bad sign. Answers that are longer than the word limit will also *not* to be awarded full marks.

  In your answer, you can use technical terms introduced in the module. For example, you can globally refer to the process of sharing information across EGP routers, computing paths and installing the corresponding entries in the forwarding table as routing *convergence*. When such a process is triggered by a network change, you can also refer to it as *reconvergence*. Similarly, you can use terms like *blackhole* for cases where a router has no forwarding entry for a specific destination, and *forwarding loop* for cases where data packets are bounced back and forth among some routers.

  From a technical perspective, you should assume that:

  - the baseline solution is exactly the same used to generate the traces in `baseline-traces/`;
  - the baseline solution updates the forwarding table only in the `finalizeIteration()` method.

- **Marking scheme**. We will mark your answers separately from each other, and assign up to 4 marks for each answer. Marks will be awarded according to the technical correctness, accuracy, completeness and brevity of each answer. The same criteria will be used to award possible partial marks.

Additional notes and suggestions

We suggest that you solve the coursework progressively, focusing on the released tests one at the time. The released tests are indeed ordered (i.e., numbered) to reflect increasing levels of complexity: later tests generally require more features to be supported by your EGP. We expect that addressing tests in the order they are provided will make building your final solution more manageable.

You can use `local-tester.sh` and the other files in your coursework tarball to test your solution. We stress however that our marking scripts can implement more checks than the ones performed by `local-tester.sh` and the checkers in `lib/checkers.py`. Rather, local-tester.sh is intended to provide you with an example (in bash) of how you can automate tests for your server: feel free to adapt it to your preferred interaction mode, change it and/or extend it to support more tests.

We also stress that the *baseline solution is not optimal*: nothing prevents your implementation from achieving better revenues in one or more tests. In fact, one challenge that you can take after finishing the coursework is to try and outperform the baseline solution: can you implement a smart EGP solution that makes more (virtual) money than the COMP0023 teaching staff?

What to submit

You must submit your work on Moodle. Your submission must include:

- one plain text file, named `egp.py`, containing the Python3 source code of your EGP implementation, solving the tasks in Stage 1 and 2;
- one plain text file, named `stage3-answers.txt`, containing your answers for stage 3.

Please refer to the previous sections for the specification of assumptions, constraints and format of both `egp.py` and `stage3-answers.txt` to be submitted.

Intended Learning Outcomes

The coursework provides a first-hand experience in the design and implementation of a (simplified) routing protocol intended to handle the peculiarities of inter-domain path computation. As such, it aims to deepen the understanding of problems, mechanisms and subtleties concerning (inter-domain) routing in networked systems, and to critically rethink them.

The coursework is fully aligned with the COMP0023 intended leaning outcome of understanding the nuances of interactions between a network's distributed entities – obviously, the distributed entities considered here are the simulated routers which needs to compute paths in a distributed fashion.

Academic Honesty

You are permitted to discuss the content of lectures and assigned readings with your classmates, but you are *not permitted* to share details of the assignment, show your code (in whole or in part) to any other student, or to contribute any lines of code to any other student's solution.

All code and answers that you submit must be the product of work done by you alone.

Copying of code from student to student is a serious infraction; it will result in automatic awarding of zero marks to all students involved, and is viewed by the UCL administration as cheating under the regulations concerning Examination Irregularities.

We use copying detection software that exhaustively compares code submitted by all students from this year's class and past years' classes, and produces color-coded copies of students' submissions, showing exactly which parts of pairs of submissions are highly similar. Do not copy code from anyone, either in the current year, or from a past year of the class.