GTNA 2.0 - A Framework for Rapid Prototyping and Evaluation of Routing Algorithms

Benjamin Schiller and Thorsten Strufe
P2P Networks - TU Darmstadt
[schiller, strufe][at]cs.tu-darmstadt.de

Keywords: Networks, Simulation, Routing Protocols, Performance Evaluation

Abstract

Routing in complex networks is increasingly optimized towards situation and properties of the underlying network. Quick hypothesis testing with respect to the performance of different strategies, however, is posing to be an unnecessarily complicated task. To this end we propose *GTNA-2*, the enhanced second version of Graph-Theoretic Network Analyzer. Based on the broadly used *GTNA*, it allows both for the efficient and simple analysis of a large set of graph metrics, but additionally has been extended with support for rapid prototyping and quick evaluation of arbitrary routing algorithms. In this paper, we discuss the implementation and evaluation of routing algorithms in *GTNA-2*. As a proof of concept, we demonstrate the framework's ease of use by comparing the routing performance of Named data Networking with basic IP-based routing.

1. INTRODUCTION

Several classes of networks with quite different properties, like for instance biological-, social-, and even linguistic networks have recently been in the focus of academic research [26, 22, 21]. A common research question of such studies is the task to find paths between two participating entities, in short: nodes, which frequently is performed with extreme adaptation to the properties of the underlying network, and quite frequently performed in a decentralized fashion, based on local knowledge only. Recent examples include routing in social networks [17], Peer-to-Peer systems [2], delay tolerant networks [11], or named data networking [12].

Analyzing the performance of different strategies for routing, together with the analysis of the underlying network, has traditionally come with quite some overhead: Formal approaches frequently require an oversimplification of the situation rendering a meaningful analysis impossible in most cases. Simulation, or measurement in the wild, yield more realistic results but involve the establishment of rather complicated simulation models, or even the the implementation of client software. Executing experiments is costly, and measuring graph metrics of the underlying, as well as the evolving networks, requires extensive logging and an additional evaluation process.

Looking at both extremes, we postulate the need for a methodology that allows for rapid prototyping and efficient performance evaluation of routing algorithms. It should allow for the simple implementation of routing algorithms, and their test on various underlying network topologies of different kinds. The networks have to be adaptable, to allow for combined routing and topology control, or automatic configuration. Direct evaluation of the properties of the underlying and evolving graphs has to be included, to facilitate a comprehensive insight into the environment and corresponding performance of the analyzed routing algorithms.

A good framework for enabling the rapid prototyping and evaluation of routing algorithm should be designed in a way, so that the implementation of a new routing algorithm and related data structures is simple and straight forward without the need to create a complete and highly complex simulation system. It should allow to easily compare different scenarios, parameters, and approaches and quickly perform evaluations to get a first impression of the new routing strategy. Without any overhead, aggregations of multiple runs should be generated and a visual representation of the results should be generated in order to quickly grasp differences between compared algorithms.

In this light, we propose the Graph-Theoretic Network Analyzer - version 2, *GTNA-2*, implemented in Java. It provides a framework for the effortless implementation of routing algorithms, as well as their analysis in various scenarios, on imported or generated network topologies. *GTNA-2* does not aim at evaluating on a high level of detail, like, packet-level simulators do. It is not designed with a specific scenario or application in mind, as it is the case for special purpose simulators. Also, it does not consider delays between nodes and is limited to static network topologies, hence, cannot be used to investigate churn. It is rather designed for a fast implementation of new ideas and to quickly generate an overview of the performance of different strategies on various topologies.

So far, a large set of routing algorithms has been implemented. This includes variations of greedy strategies, different versions of lookahead approaches, multi-phase algorithms, random walks, and approaches based on routing tables. In combination with many identifier spaces and network topology generators, all implemented and ready-to-use, a newly implemented routing algorithm can be compared to a plethora of other approaches in diverse scenarios.

Having used *GTNA-2* for several studies on Peer-to-Peer, Darknet, and Wireless Sensor Network routing in the past, we apply it to the task of prototyping and analyzing routing in Named Data Networking, as it has broadly and frequently

been discussed in the recent past. We show that implementing even such a comparably complex system is simple to evaluate using *GTNA-2*, and provide initial insights into the performance of the prototyped system.

2. BACKGROUND

In this Section, we describe a generalized understanding of routing in distributed systems. Then, we describe the main components of the proposed framework, *GTNA-2*, and discuss related work in the area of simulating distributed systems.

2.1. Routing in distributed systems

In distributed systems, nodes provide services that can be requested by others. Such services include the lookup, storage, and update of data like web pages, video streams, measurements, or profile information. The main challenge in systems, such as Wireless Sensor Networks [1], Peer-to-Peer networks [24, 15], Friend-to-Friend networks [6], Named Data Networks [12], or Delay Tolerant Networks [11, 14], is to find the provider of a specified service. Due to the distributed nature of these systems, no central knowledge about the location of such services exists.

Upon the receipt of a service request, either from the local user or a neighbor, a node checks if the service can be served directly with local resources. In this case, a response is created and the request hence terminated successfully with a reply, i.e., the requested resource. Otherwise, the request is delegated to another node. To choose on of its neighbors as target for the next hop, the node applies a local routing algorithm, based on information about the request, its neighbors, and context information about the structure of the overall topology of the network. If no such neighbor can be found, the request fails because the service could not be provided.

A common example for routing in distributed systems is the retrieval of data stored in a distributed hash table like Chord [24]. A request contains an identifier from the Chord identifier space $[0, 2^p - 1]$ to identify the data item that should be retrieved. Peers that receive such a request first check if they have the original copy or a replica of the requested data item in their local data storage. If this is the case they can immediately answer the request. Otherwise, they apply a greedy routing algorithm by forwarding the request to their neighbor whose identifier is closest to the target identifier of the requested data item.

2.2. Framework Components

In previous work, we have presented the *Graph-Theoretic Network Analyzer* (*GTNA*) [20]. *GTNA* is a highly extensible Java-based framework, aimed at the graph-theoretical analysis of network topologies. These topologies can either be read from existing files, like, e.g., network snapshots or crawled graphs, or generated using the *Network* component (cf. Figure 1). The *Graph* component stores and represents these simple

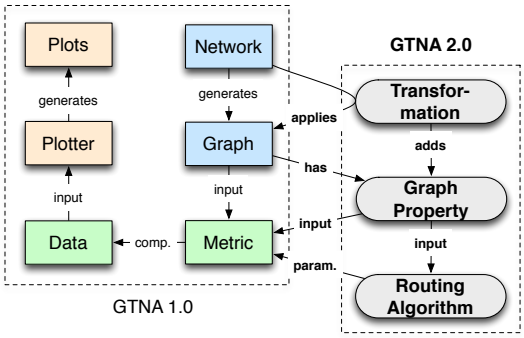

Figure 1: Components of GTNA

directed graphs. It serves as input for the *Metric* component which computes multi- and single-scalar metrics from different areas depending on the input graph. The computed metrics are processed and stored in the *Data* component which is responsible for combining and representing results from multiple runs. The *Plotter* components allows for the generation of various plots to visualize the results generated by the *Metric* component.

Since its first release, the framework has been extended with three main components: *Transformation*, *Graph Property*, and *Routing Algorithm*. Graph properties provide additional information about a graph, its nodes, or edges. During generation, networks can add such properties to express node weights, the locations of nodes on a map, community structures, and similar information. Transformations are applied after generating a graph using a network. They can either change the structure of the given graph by adding or removing nodes and edges or append new graph properties to the existing graph. A graph and its assigned properties are used as input for the computation of metrics. Graph properties are also used by routing algorithms which serve as a parameter for the routing metrics that evaluate them in a specified scenario.

A detailed description of the new components and how they are used to enable the evaluation of routing algorithms in *GTNA-2* is described in Section 4.1..

2.3. Related work

The evaluation of routing algorithms using network simulation is a common approach. Network simulators like OMNeT++ [25] and NS-3 [9] allow for a detailed modeling of message flows including underlay topologies, delays between hosts, and congestions. While complex simulation models achieve realistic results, they are complicated to implement and debug. Due to the level of detail that can be modeled using these tools, they also impose a high overhead for the

evaluation and therefore are not well-suited for the rapid prototyping and testing of routing algorithms.

In order to reduce the time and effort of implementing and evaluating distributed systems in complex network simulators, many special purpose simulators have been developed. Frameworks for the simulation of Peer-to-Peer networks like OverSim [5], PeerfactSim [23], and PlanetSim [7] commonly create three abstraction layers to model complex network structures: network, overlay, and application. Tools for the evaluation of applications in the area of Wireless Sensor Networks like SIDnet SWANS [8] add support the simulation of wireless communication. ONE [13], a simulation framework designed for the evaluation of protocols in Delay Tolerant Networks, provides an abstraction of the changing network connectivity between nodes based on realistic movement models. These special purpose simulation frameworks are well-suited for an analysis of systems in their respective area as they are already tailored to the specific requirements and application scenarios. But they are not well-suited for the use in other scenarios and therefore cannot be applied as tools for the rapid evaluation of routing algorithms in general.

Neither network simulators nor special purpose frameworks provide easy means to compute graph-theoretic properties on the network topologies generated during a simulation run. Computing such metrics often is a cumbersome task, requiring the adaptation of measured data and extraction from log files. Another possibility is the use of additional analysis tools, such as Gephi [3], Pajek [4], or Jung [16].

In clear contrast, the routing components of *GTNA 2.0* are easy and fast to implement and allow for a rapid evaluation of new routing algorithms. Comparing different routing algorithms or parametrized versions is easy. Complex components, as provided by most network simulators, are left out for the benefit of a fast evaluation which is important when developing and testing new algorithms. In addition, the routing components are built on top of the graph-theoretic analysis components. They allow for the computation of graph-theoretic metrics without additional implementation overhead and therefore provide a key feature missing in common network simulation frameworks.

3. METHODOLOGY FOR ROUTING EVALUATION

When evaluating a routing algorithm in a distributed system, multiple simulation runs have to be performed to eliminate outliers. During every simulation run, the nodes in a distributed system attempt to find the provider of a specified service starting at a dedicated node in the system. Therefore, every simulation run requires two input parameters: a source and a target. The source is a node inside the network that initially issues the request, e.g., retrieving a document from a web server. The target is a specification of a service that is supposed to be provided as a result of the request, e.g., the address of the web server and the name of the document.

In every step of the simulation of a routing attempt, starting at the source itself, the routing algorithm can select between three different outcomes: *success*, *forward*, or *failure* (cf. Figure 2). In case the requested service is provided at

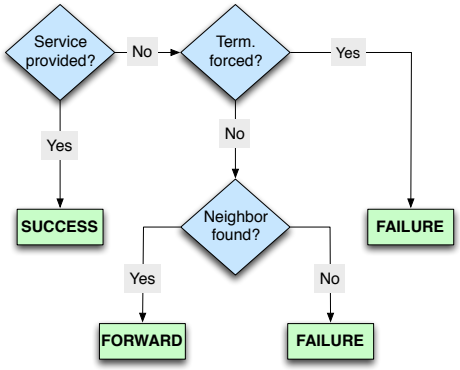

Figure 2: Decision tree of each step of a routing algorithm

the current node processing the request, the routing algorithm succeeds in finding a service provider and the simulation can be stopped. If the service is not provided at the current node, the algorithm can decide to terminate with a failure, e.g., because a time to live counter has been exceeded prohibiting a forward of the message. If this is not the case, the algorithm has to check if there exists a neighbor that the request can be forwarded to. If no such neighbor exists, the attempt to find a service provider ends with a failure. Otherwise, the message is forwarded to the selected neighbor and the process is repeated.

Besides some special break conditions, the main difference between routing algorithms is the selection of the next neighbor to forward a request to. It depends on the specified target, the information a node has about its neighbors and the network, as well as additional information given as part of the request. In the case of distributed hash tables like Chord [24], every node has information about the location of its neighbors in the identifier space. As the target of a routing request is determined by an identifier from the identifier space, a node computes the distances between its neighbors' identifiers and the target identifier and forwards the message to the neighbor with the shortest distance. Another example is IP-based routing. Here, the target is specified by the IP address of the destination host. When receiving a request, a node looks up the longest matching prefix of the target address in its routing table and forwards the request to the respective neighbor. Additional information can be a time to live counter that indicates how often a request is supposed to be forwarded before being discarded.

The overall workflow of every simulation run is depicted in Figure 3. For every run, a new source S and target T are selected from the set of all nodes in the system which serve

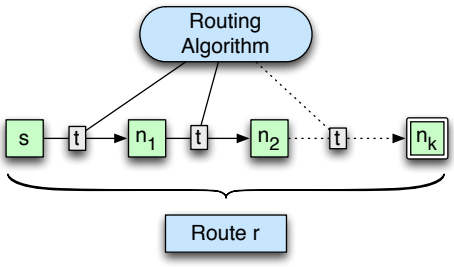

Figure 3: Methodology for evaluating a routing algorithm

as input for the execution of the run. The routing algorithm is then applied to the source node S to determine a neighbor N_1 to which the request for the target resource T should be forwarded. Then, the request is sent to N_1 and the routing algorithm is applied to determine the next node to forward the request to. After k iterations, the request delegation terminates at a node N_k. In case N_k can provide the requested service, the simulation run is terminated successfully. Otherwise, it terminates with a failure indicating that a termination was forced or no further neighbor could be found to forward the request to.

As a result of such a simulation run, we consider the outcome, i.e., *failure* or *success*, as well as the route taken by the request, i.e., the sequence $R = \{S, N_1, N_2, \ldots N_k\}$ of nodes that have seen the request.

4. REALIZING THE METHODOLOGY

In this Section, we describe three of the components that have been added to since its first publication and are relevant for the routing evaluation: graph properties, transformations, and routing algorithms. Then, we discuss how the methodology, as described in Section 3., is realized in *GTNA-2* using these new components.

4.1. Graph Properties

In *GTNA*, data structures for storing graphs only represent nodes and edges. In order to store information like, e.g., edge or node weights, a community structure of the network's topology, or node colors for visualization, such information can be expressed as a graph property and added to a graph.

In this Subsection, we describe three types of graph properties which are closely related to the evaluation of routing algorithms: *IdentifierSpace*, *DataStorageList*, and *RoutingTables*. These properties are commonly used by routing algorithms to determine if a node can provide a requested service and to decide which neighbor to forward a request to.

4.1.1. IdentifierSpace

As described in Section3., a request which is processed in a distributed system contains a target that describes how to successfully answer the request. This target is an element from what is usually referred to as address space, namespace, or identifier space. This includes IP addresses, document names, and identifiers of a distributed hash table. Without loss of generality, we use the interchangeable term identifier space from now on.

In *GTNA-2*, an identifier space is logically divided into a set of partitions each of which is assigned to a node in the network. Commonly, the union of all partitions represents the whole identifier space and any two partitions are disjoint regarding the set of contained identifiers. But this is no requirement enforced by *GTNA-2* or any of its components.

Given two identifiers, a partition can decide which one is considered to be closer with respect do an understanding of closeness. Also, a partition can determine if a specific identifier is contained in itself or not.

Analogously, an identifier can also decide which of two given identifiers or partitions is considered closer. Given a set of nodes, it can also determine the node whose partition is closest to itself.

While providing a notion of closeness between partitions and identifiers is sufficient for making greedy routing decisions in most cased, some algorithms require the possibility to explicitly compute the distances between partitions and identifiers. Identifier spaces that provide this possibility enable the implementation and evaluation of routing algorithms that include the explicit distance between nodes in their routing decision, e.g., to calculate weights for probabilistic routing decisions.

A simple example of an identifier space is a *bounded number line* that covers the interval $[0,1)$. Partitions are intervals $p_{a,b} := [a,b)$ with $0 \leq a < b \leq 1$. Identifiers are all numbers inside the bounded number line, i.e., $id \in [0,1)$. Distances can be computed in different ways, the simplest one being the absolute distance between the two identifiers: $d(id_1, id_2) = |id_1 - id_2|$.

So far, a large number of different identifier spaces has been implemented in *GTNA-2*. This includes variations of d-dimensional ring identifier spaces with different distance metrics, e.g., hyperbolic, maximum norm, euclidean, and Manhattan. Together with the respective network generator, identifier spaces for well-known distributed hash tables have been implemented such as Chord [24] and Kademlia [15].

4.1.2. DataStoreList

Often, it is sufficient to test the capabilities of routing algorithms by terminating at a node whose partition contains the target identifier and therefore is assumed to be responsible for providing the respective services. In other cases, it can be desirable to place replicas of the considered services in the distributed system.

This allows for an evaluation of the impact of replication on the success of requests and the number of hops required to find the target services.

In general, replication of services is used to speed up the handling of application layer requests while the replication itself can be done on different layers. Depending on the layer and kind of information that is replicated in the system, different terms like data store, cache, and registry are used to describe the data structure used for storing these replicas. Without loss of generality, we use the term data store from now on to refer to a data structure that allows nodes in a distributed system to store replicas of the provided service. Analogously, we refer to the replicated services stored in the data store as data items.

The graph property *DataStoreList* is used to represent the data stores of all nodes in a network. Each data store contains information about the data items that are stored at the respective node. The data store distinguishes between source and replica data items, i.e., data items which originate at a node in contrast to replicas stored for other nodes. This differentiation has implications for data stores that can only store a fixed number of data items. In such a case, only replicas are supposed to be deleted while source data should always be preserved when adding new data to it.

So far, three different data stores have been implemented: *UnlimitedDataStore*, *FifoDataStore*, and *LruDataStore*. The *UnlimitedDataStore* allows the storage of an unlimited number of replica data items. In the *FifoDataStore*, replicas are stored in a queue and the oldest element is removed when the maximum queue size is reached. Similarly, the *LruDataStore* removes the least recently used element from the replica storage in case its maximum size has been reached.

4.1.3. RoutingTables

The third type of graph properties relevant for routing in *GTNA-2* are routing tables. Given the target identifier of a request, a routing table determines the next hop, the request should take. In contrast to using distances computed on the basis of identifier spaces, routing tables can easily be precomputed. Also, they directly reflect routing tables as they are used in IP-based routing on the Internet and similar prefix-based routing schemes.

4.2. Transformations

Transformations are used to change the structure of a graph or add information to it. They are applied after the generation of a graph using the *Network* component.

Changing the structure of a graph means to add or remove nodes or edges from it. A simple example is the *Bidirectional* transformation. It transforms any given graph into a bidirectional graph by adding the inverse edge (b,a) for every edge (a,b) in case it does not exist already. The *UnitDiscGraph* requires the given graph to be equipped with a 2-dimensional identifier space. All existing edges are removed from the graph and new ones added for all nodes whose euclidean distance is smaller than or equal to a given unit.

Adding information means to add a graph property to an existing graph. An example for such a transformation is *CDLPA*, an implementation of the Label Propagation community detection algorithm proposed by Raghavan et al. [18]. The transformation adds a new graph property to the graph that describes its community structure. Similarly, various graph drawing algorithms are implemented as transformations that add 1- or 2-dimensional identifier spaces to any given graph.

4.3. Routing Algorithms

Routing algorithms in *GTNA-2* are implemented according to the methodology as described in Section 3.. When executed, a routing algorithm has access to the corresponding network's graph and its properties, e.g., identifier space and data storage. Given a start node that issues a request and a target identifier, a routing algorithm returns the generated route. It consist of a list of nodes in the order in which they have seen the request and a flag indicating if the request could be served by the last node or not.

In case a data store is available as a graph property, a node is assumed to be able to serve a request if its data store contains a data item for the target identifier. If only an identifier space is available, a node is assumed to serve a request successfully in case the target identifier is contained in its partition.

So far, *GTNA-2* contains a large number of routing algorithms. Besides many variations of distance-based greedy routing algorithms, different versions of lookahead algorithms as well as routing based on routing tables are included. Experimental algorithms allow the obfuscation of identifier or the combination of multiple algorithms during different phases of the request delegation.

4.4. Evaluation of routing algorithms

The evaluation of routing algorithms is implemented as a metric in *GTNA-2*. Thereby, the evaluation of routing algorithms is integrated in the framework's regular workflow and takes advantage of the existing infrastructure regarding data aggregation and result visualization.

The routing metric takes three parameters: a routing algorithm, a source selection algorithms, and a target selection algorithm. In every simulation run, the source selection algorithm determines a new node that issues the request in this run. The target identifier for this request is computed using the target selection algorithm. Both parameters are given to the routing algorithm that computes a route depending on the network topology, the contained graph properties, the source node, and the target identifier. Based on the computed routes of all simulation runs, various metrics like the hop count distribution, average path length, and success rate are computed.

5. GTNA IN USE

In this Section, we give an example how routing algorithms can be implemented, compared, and evaluated in *GTNA-2*. We give two examples for the implementation of routing algorithms and show how simple simulations can be executed using the *GTNA-2* framework.

On the Autonomous System level of the Internet, routing decisions are based on longest-prefix matches in routing tables built using the Border Gateway Protocol [1]. Whenever a node receives a request, it checks its routing table and forwards the request to the node that is listed as the next hop for the given IP address. List 1 gives an excerpt of the implementation of a single routing step for a routing algorithm based on routing tables.

```
1 // add current node to route
2 route.add(current);
3 // terminate if service can be provided
4 if (isEndPoint(current, target))
5         return new Route(route, true);
6 // terminate with a failure in case ttl exceeded
7 if (route.size() > ttl)
8         return new Route(route, false);
9 // failure if no next hop can be determined
10 int nextHop = getRoutingTable(current).getNextHop(target);
11 if (nextHop == current || nextHop == RoutingTables.noRoute)
12         return new Route(route, false);
13 // forward request to next hop according to routing table
14 return this.routeToTarget(route, nextHop, target, nodes);
```

Listing 1: Implementation of a routing for IP-based routing

After adding the current node to the route object (line 2), the routing algorithm checks if the node is an end point for the request, i.e., it can serve the request for the target identifier (line 4). If this is the case, the request delegation is successfully terminated (cf. decision tree in Figure 2). In case the time to live counter is exceeded (line 7), the algorithms exits with a failure. Lastly, a lookup for the next hop in the routing table is performed (line 10). If the retrieved destination is valid, the request is forwarded to the corresponding neighbor. Otherwise, the routing is terminated and the returned route marked as failed.

In [12], Jacobson et al. propose the new paradigm of *Named Data Networks* (NDN) for requesting and delivering data over the Internet. Instead of always routing requests for data items to a specific destination, requests are forwarded based on the name of the requested content. Retrieved data is cached on the way back to the requesting node and thereby replicated in the network. A very simple realization of this principle is to use an IP-based routing algorithm and simply replicate data on the inverse path of a successful request. An implementation in *GTNA* which is a direct child class of the IP-based routing algorithm described above is shown in Listing 2.

```
1 // perform IP-based routing
2 Route route = super.routeToTarget(start, target);
3 // replicate data on the whole route if successful
4 if (route.isSuccessful())
5         for (int node : route.getRoute())
6                 getStorageForNode(node).addReplica(target);
7 return route;
```

Listing 2: Implementation of a step for NDN routing

First, the request is processed using the regular routing algorithm (line 2). In case the request terminates successfully, the requested data item is replicated on all nodes on the path between requester and service provider (lines 4-6). Since over time, more and more data items are replicated in the network, the probability increases that a request hits a replica and terminates with less hops than it would take to reach the initial source of the data item.

In Listing 3, we give an example how these two routing algorithms can be evaluated and compared using *GTNA-2*.

```
1 // applied transformations
2 Transformation t0, t1, t2, t3;
3 t1 = new NodeIds();
4 t2 = new NodeIdsRoutingTable();
5 t3 = new NodeIdsDataStorage(new LruDataStore(0, 30), 1);
6 Transformation[] t = new Transformation[] { t1, t2, t3 };
7 // creating network instances
8 Network[] nw = new Network[filenames.length];
9 for (int i = 0; i < filenames.length; i++)
10         new ReadableFile("AS", "caida", filenames[i], t);
11 // creating metric instances
12 SourceSelection ss = new ConsecutiveSourceSelection();
13 TargetSelect ts = new DataStorageRandomTargetSelection();
14 Metric r1 = new Routing(new IpRouting(), ss, ts);
15 Metric r2 = new Routing(new NdnRouting(), ss, ts);
16 Metric ds1 = new DataStorageMetric();
17 Metric ds2 = new DataStorageMetric();
18 Metric dd = new DegreeDistribution();
19 Metric[] metrics = new Metric[] { r1, ds1, r2, ds2, dd };
20 // generating networks, simulate routing, compute metrics
21 Series[] s = Series.generate(nw, metrics, 100);
22 // plot the results
23 Plotting.multi(s, metrics, "multi/");
24 Plotting.single(s, metrics, "single/");
```

Listing 3: Code for executing simulations of IP and NDN routing

Since we attempt to evaluate routing between Autonomous Systems on the Internet, we use the AS topology provided by the CAIDA project [10] [2]. Various snapshots of the AS topology between 2007 and 2013 are read using the *ReadableFile* network generator (lines 8-10). Since the provided graph does not contain any additional information, we utilize transformations to add an identifier space similar to IP addresses (line 3), a routing table based on it (line 4), and a data store with a Least Recently Used replacement strategy, a maximum size of 30 replicas, and one random data item assigned as source to each node (line 5). In the following, instances of the *Routing* metric with the two different routing algorithms, *IP* and *NDN*, are created (lines 14-15). An instance of the *DegreeDistribution* metric as well as two instances of the *DataStorageMetric* are also added to the set of metrics to be computed (lines 16-19). The *DataStorageMetric* is used to compute statistics

[1] http://www.ietf.org/rfc/rfc1771.txt

[2] http://data.caida.org/datasets/topology/ipv4.allpref24-aslinks/

about the *DataStorageList* graph property, e.g., the distribution of the data storage size. In the last part, the simulation of the routing on the given network topologies is initiated (line 21) and the results plotted using the *Plotter* component (lines 23-24).

Figure 4 shows the statistics measured by the *DataStorageMetric* for the cases of pure IP-based routing and the routing using NDN. In the first case, the data store of each node only

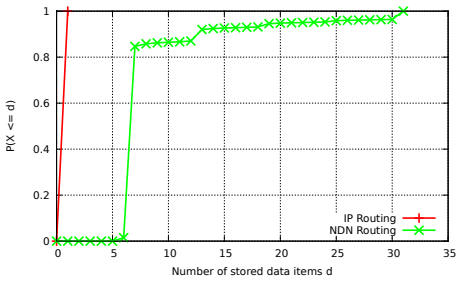

Figure 4: Storage size distribution without and with replication in NDN

contains a single element, the one this node is the source for. In the case of NDN, replicas are distributed throughout the network. With a maximum number of 30 replicas per node, the maximum total size of a node's data store is 31.

As expected, the distribution of replicas in the network has a direct impact on the hop count of data requests in the network. This is indicated by Figure 5 which shows the average hop count of the routing algorithms evaluated on the network snapshots between 2007 and 2013 plotted against the number of nodes in the network. For both routing algorithms, the av-

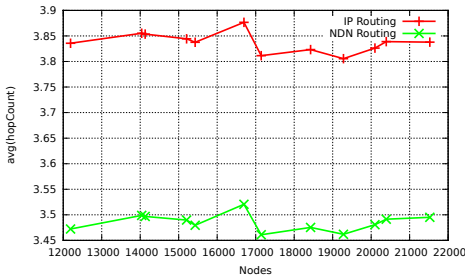

Figure 5: Average hop count of IP-based vs. NDN routing 2007 - 2013

erage hop count is roughly independent of the network size. The average decrease of 0.35 hops on average when using NDN instead of IP routing is also independent of the network size.

6. EVALUATION

In this Section, we briefly discuss the memory requirements of *GTNA-2* as well as the runtimes for simulating the routing algorithms as described in Section 5..

6.1. Memory requirements

To see how much memory is required for running simulations in *GTNA-2*, we generated random graphs of different size and number of edges. We chose the number of nodes between 2^{12} and 2^{20} and the number of edges between 0 and 2^{20}. Then, we measured the amount of memory required to store the generated data structures.

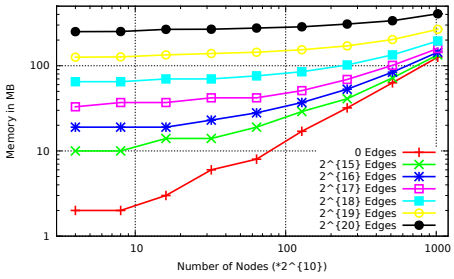

Figure 6: Memory required to represent graphs in *GTNA-2*

Figure 6 shows the memory required to store the generated graphs depending on the number of edges and nodes. The given values are the maximum value obtained for each configuration from 50 repetitions. They indicate, that the space required for representing a graph in *GTNA-2* grows linearly with the number of nodes and the number of edges, implying a space complexity of $O(N+E)$.

Without any edges, a graph containing 2^{20} nodes requires $126MB$ of space which implies a storage footprint of $\approx 0.123KB$ per node. Adding 2^{20} edges to the graph increases the total memory usage to $410MB$, implying an average footprint of $\approx 0.277KB$ per edge.

These measurements show, that *GTNA-2* is in fact capable of representing large networks and hence can be used for large-scale simulations and evaluations of routing algorithms. In comparison, Baumgart et al. report a memory footprint of $35KB$ per node in OverSim and P2PSim which is much higher than the footprint of *GTNA-2* [5]. They also note, that the used memory in both systems grows proportionally with the number of nodes. Since the memory requirements of graphs in *GTNA-2* grows linearly but subproportional it is much more suited for the representation an evaluation of sparse networks since the number of edges has a much higher impact on the memory footprint.

6.2. Runtime

In the comparison of IP- and NDN-based routing approaches (cf. Section 5.), we evaluated the routing on 12 snapshots of the AS-level Internet topology obtained by the CAIDA project. The selected snapshots represent the Internet roughly every 6 months with the number of nodes growing from 12k to over 21k. The number of edges increases from 25k to over 45k.

Table 1: Average simulation duration per run

Snapshot	Nodes	Edges	IP	NDN	Routing Tables
2007-09-13	12,190	25,822	.28 sec	.49 sec	21.23 sec
2008-01-02	14,038	28,714	.29 sec	.52 sec	29.78 sec
2008-07-02	14,128	29,218	.29 sec	.52 sec	29.17 sec
2009-01-03	15,205	31,328	.32 sec	.57 sec	35.15 sec
2009-07-02	15,427	31,706	.31 sec	.57 sec	34.24 sec
2010-01-01	16,695	33,315	.35 sec	.63 sec	41.87 sec
2010-07-01	17,143	35,765	.36 sec	.64 sec	45.77 sec
2011-01-02	18,427	39,642	.40 sec	.69 sec	52.93 sec
2011-07-01	19,281	41,131	.40 sec	.71 sec	60.31 sec
2012-01-02	20,098	42,338	.45 sec	.82 sec	71.06 sec
2012-07-03	20,393	43,097	.45 sec	.81 sec	74.46 sec
2013-01-02	21,525	45,320	.46 sec	.85 sec	82.84 sec

During the evaluation, we measured the time for the simulation of routing using IP- and the NDN-based strategy as well as the time for the generation of the routing tables for all nodes of the network graph. These measurements show, that the generation of the required routing table data structures as well as the execution of the routing simulation takes less than 90 seconds even on the largest topology investigated (cf. Table 1). Obviously, the runtime of the routing simulation highly depends on the average hop count during simulation. By default, a total of $5 * |N|$ source / target pairs are selected and the routing towards the target performed from the source node. Hence, in the default setting, the runtime of a routing simulation grows linearly with the number of nodes.

These results clearly show that *GTNA-2* is actually capable of quickly executing routing simulations. This allows researchers to quickly test hypothesis and newly developed routing algorithms without waiting a long time for the first results.

7. SUMMARY, CONCLUSION, & OUTLOOK

While network simulators allow for simulation with a high level of detail, they are often complex to implement and hard to debug. Special purpose simulation frameworks allow for a higher level of abstraction but are tailored for rather specific use-cases and therefore not easily applicable to the evaluation of arbitrary routing scenarios. In both cases, it is rather complicated to collect information about the global network structure and compute graph-theoretic metrics of the generated network topologies.

In this paper, we introduced *GTNA-2*, a framework for rapid prototyping and evaluation of routing algorithms. It closes the gap between network simulators and graph analysis tools. *GTNA-2* offers an easy-to-use interface for implementing routing algorithms. In the evaluation, we have shown that *GTNA-2* has a comparably small memory footprint and allows for a quick evaluation and comparison of different routing schemes. In addition to the simulation of routing on stativ topologies, *GTNA-2* takes full advantage of the underlying graph theoretical analysis framework allowing a graph-theoretic analysis of the generated and investigated topologies at the same time.

In the future, we plan to extend *GTNA-2* to allow for the computation of metrics on evolving networks. We are also investigating the possibility to integrate interfaces to graph databases and to extend the already existing multi-threading capabilities for the evaluation of routing algorithms. Furthermore, we are planing to support simulation of parallel routing requests in order to correctly model routing as proposed for Peer-to-Peer systems like Kademlia.

REFERENCES

[1] Al-Karaki and Kamal. Routing techniques in wireless sensor networks: a survey. *Wireless Communications, IEEE*, 2004.

[2] Balakrishnan et al. Looking up data in p2p systems. *Communications of the ACM*, 2003.

[3] Bastian et al. Gephi: An open source software for exploring and manipulating networks. 2009.

[4] Batagelj and Mrvar. Pajek - program for large network analysis. *Connections*, 1998.

[5] Baumgart et al. Oversim: A flexible overlay network simulation framework. In *IEEE Global Internet Symposium*, 2007.

[6] Clarke et al. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, 2001.

[7] García et al. Planetsim: A new overlay network simulation framework. *Software Engineering and Middleware*, 2005.

[8] Ghica et al. Sidnet-swans: A simulator and integrated development platform for sensor networks applications. In *Embedded network sensor systems*, 2008.

[9] Henderson et al. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 2008.

[10] Huffaker et al. Topology discovery by active probing. In *Applications and the Internet (SAINT) Workshops*, 2002.

[11] Hui et al. Bubble rap: Social-based forwarding in delay-tolerant networks. *Transactions on Mobile Computing*, 2011.

[12] Jacobson et al. Networking named content. In *Emerging networking experiments and technologies*, 2009.

[13] Keränen et al. The one simulator for dtn protocol evaluation. In *Simulation Tools and Techniques*, 2009.

[14] Leguay et al. Dtn routing in a mobility pattern space. In *Delay-tolerant networking*, 2005.

[15] Maymounkov and Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. *Peer-to-Peer Systems*, 2002.

[16] O'Madadhain et al. Analysis and visualization of network data using jung. *Journal of Statistical Software*, 2005.

[17] I. Parris and T. Henderson. Privacy-enhanced social-network routing. *Computer Communications*, 2012.

[18] Raghavan et al. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 2007.

[19] Schiller et al. Gtna: A framework for the graph-theoretic network analysis. In *Spring Simulation Multiconference*, 2010.

[20] Scholand et al. Social language network analysis. In *Computer supported cooperative work*, 2010.

[21] Sharma et al. Inferring who-is-who in the twitter social network. In *Workshop on Online Social Networks*, 2012.

[22] Stingl et al. Peerfactsim.kom: A simulation framework for peer-to-peer systems. In *High Performance Computing and Simulation*, 2011.

[23] Stoica et al. Chord: A scalable peer-to-peer lookup service for internet applications. *Computer Communication Review*, 2001.

[24] Varga et al. The omnet++ discrete event simulation system. In *European Simulation Multiconference*, 2001.

[25] Wong et al. Biological network motif detection: principles and practice. *Briefings in Bioinformatics*, 2012.