Simulating Large-scale Dynamic Random Graphs in OMNeT++

Kristján Valur Jónsson[*]
School of Computer Science
Reykjavik University, Iceland
kristjanvj@ru.is

Ýmir Vigfússon
School of Computer Science
Reykjavik University, Iceland
ymir@ru.is

Ólafur Helgason
Laboratory for Comm. Netw.
Electrical Engineering
KTH - Royal Inst. of Tech.
olafur.helgason@ee.kth.se

ABSTRACT

Simulating large-scale dynamic systems becomes increasingly more important as real-world systems grow in scale. We present a set of components for the OMNeT++ discrete event simulator which enable efficient modeling of large-scale random graphs that capture real-world properties, e.g. scale-free networks and small-world topologies. The complexities of the network are abstracted into a single component while enabling detailed end-system modeling in a realistic connectivity graph. We describe the modeling components and demonstrate their use in a case study of a distributed aggregation protocol.

Categories and Subject Descriptors

I.6.7 [**Simulation and Modeling**]: Simulation Support Systems—*Environments*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Discrete event*; C.2.4 [**Distributed Systems**]: Distributed applications

General Terms

Simulation, Algorithms, Design

Keywords

Discrete event simulation, OMNeT++, random graphs, distributed aggregation protocols

1. INTRODUCTION

Communications systems are growing in size and complexity – from the relatively modest sized and well-defined enterprise networks to newer paradigms, such as, ad-hoc wireless

---

sensor networks [27], wireless handheld devices [17] and the Internet of Things [10]. Evaluating new protocols for such large distributed systems is a challenging task. A realistic simulation model is an essential tool since large-scale experimental platforms for research are rare, and invariably exclusive and expensive. Simulations provide the protocol designer with the means to conduct repeatable trials to evaluate and compare protocols and fine-tune parameters.

The performance of distributed systems is frequently dependent on their execution environment. Some systems, such as wireless sensor networks, depend on channel characteristics. Others, for example VANETs [16], are characterized by mobility. In general, a realistic network *graph* representing the real-world scenario must be constructed to evaluate new protocols. However, the complex nature of many large-scale networked systems often precludes construction of fine-grained replication of real-world execution environments.

Rather than attempting to produce a fine-grained model, such as pedestrian mobility within a specific city region [17] or vehicles traveling along a specific grid [1], we may want to ask more general questions, for example regarding protocol performance as a function of churn rate, given some general system assumptions. Dynamic random graphs with carefully chosen parameters can be used to model such systems, abstracting from the complexity of the underlying network and thus allowing the protocol designer to focus on the end-system.

In this paper, we present a set of components for the OMNeT++ discrete event simulator [22] which provide simple and efficient simulations of complex distributed systems over random graphs. Dynamic lifetime management of simulated nodes is provided by a `nodeFactory` component, while a `topologyControl` component manages the graph. The topology manager uses a plug-in based graph generator architecture which provides flexibility in configuration as well as simple implementation and management of random graph algorithms. We demonstrate the application of the components by modeling the GAP [4] distributed aggregation protocol in a dynamic random graph topology.

2. BACKGROUND AND RELATED WORK

Random Graphs

Scientists have long been fascinated by networks generated from human interactions, such as social interactions (Facebook) or network peering relationships (the Internet). The
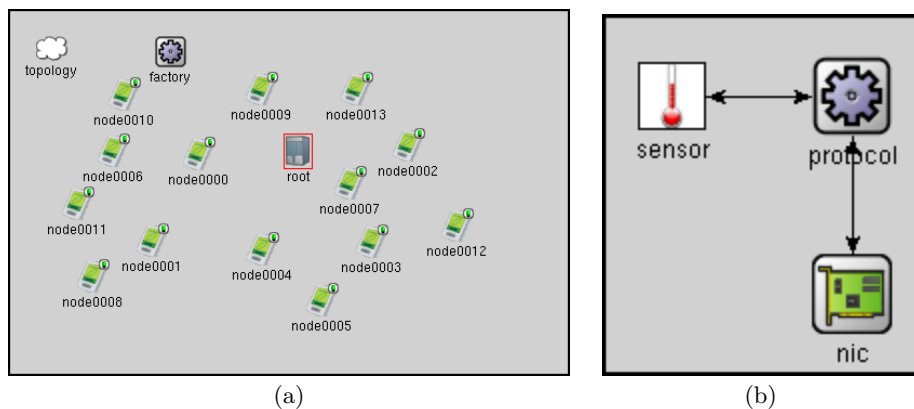
Figure 1: (a) An example simulation scenario. (b) The components of a modelled node in the scenario.

first models of large-scale behavior embedded in such real-world networks were based on the pioneering work on random graphs by Erdös and Rényi (ER) [6], but were later shown to be too limited. In particular, Watts and Strogatz [23] noted that real-world networks tend to have large clustering coefficients yet short paths between pairs of nodes on average (logarithmic in the number of nodes), whereas the ER model produces a uniformly low clustering coefficient. Graphs with this property are commonly called *small-world networks*, inspired by Milgram's experiment showing six degrees of separation between people [19]. Extensive follow-up work has shown that small worlds are common, with diverse examples that include the World Wide Web [2], metabolic pathways of worms [23], many social networks [15] and electrical grid systems [23]. For instance, Leskovec and Horvitz [15] showed that the average path length among contacts in Microsoft Messenger is 6.6.

In addition to being small worlds, many real-world networks have also been shown to exhibit an asymptotic power-law degree distribution, meaning that the probability $P(k)$ of a node having k neighbors decays as $P(k) \propto k^{-\gamma}$ for some constant γ [3], usually $2 < \gamma < 3$. Networks of this type are also known as *scale-free networks* and nodes of high-degree (substantially larger than the average) are called *hubs*. Recently, the scale-free properties of several types of networks are being debated [25]. The most prominent model to explain the emergence of scale-free networks is the *preferential-attachment* or "rich get richer" model, where nodes are introduced sequentially and connect to some existing node with probability proportional to its in-degree.

Simulation and graph generation

Several simulation tools have addressed the issue of generating and maintaining random graphs, scale-free networks and small-world topologies. PeerSim [20] is a widely used simulator for evaluating large-scale peer-to-peer systems. PeerSim supports node churn and allows for creating, and dynamically updating, different types of overlay topologies [12]. PeerSim is however based on Java and thus not compatible with OMNeT++.

GT-ITM [9], Brite [18] and Inet [26] are topology generators that can create topologies whose graphs show a power law distribution in node degree. These generators can be used to create topologies that can be used with OMNeT++

(either by direct export or via intermediate converters). These generators are however mainly useful for generating static topologies and do not easily support churn and dynamically evolving graphs. Similarly, the ReaSE framwork [8] for OMNeT++ implements a topology generation algorithm that can create a static graph, both at the AS level as well as the router level.

We are not aware of any components for OMNeT++ which generate and maintain dynamic random graphs at runtime.

3. THE SIMULATION TOOLBOX

We present a set of components[1] for OMNeT++ for simple and efficient modeling of communications patterns between nodes, whose interactions can be modeled by a random graph. We approach the network modeling abstractly, rather than specifically as in the case of modeling channels and mobility. In fact, the components can be used to model more abstract concepts such as social networks.

An example scenario which uses our toolbox is shown in Figure 1(a). The `factory` dynamically instantiates and destroys nodes (`node*` in the scenario) using user configurable probability distributions, while `topology` manages a dynamically evolving virtual communications graph. Hosts communicate using a special NIC component, as shown in Figure 1(b), which registers generated nodes as members of the topology and makes them aware of reachable neighbors.

3.1 Dynamic Node Generation

Dynamic node generation and destruction is handled by the `NodeFactory` component, implemented as a OMNeT++ simple module. This component is based on our earlier work in the *Opposim* project [11]. The factory takes the following parameters:

nodeType: The class name of a OMNeT++ compound module – the type of node to be manufactured.
generateInterval: The node generation interval.
lifetime: The lifetime of generated nodes.
minLifetime: The minimum lifetime of generated nodes.

Node generation events are scheduled according to the generateInterval. This is a volatile parameter, allowing any

---

[1]The components are available at `https://github.com/kristjanvj/DRGSimLib` and released under a open source license.
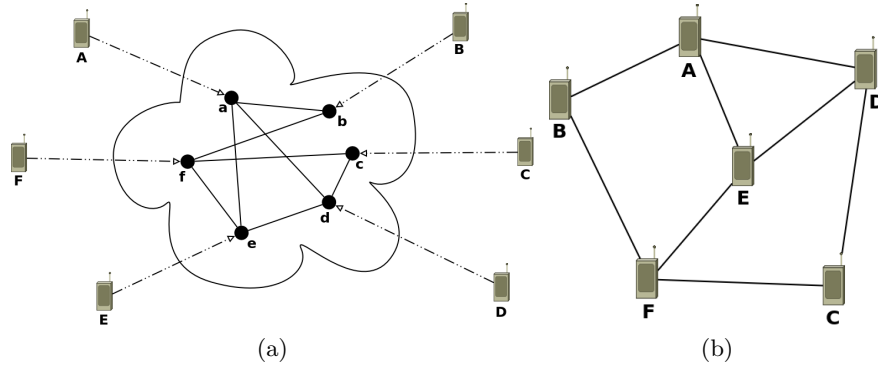
Figure 2: Modeled network. (a) The nodes represented by upper case letters indicate simulated systems, while lower case nodes indicate the corresponding vertices $v \in G^t = (V^t, E^t)$ – the data structure maintained internal to the topologyControl instance. (b) An isomorphic representation of the communications network at time t.

of the OMNeT++ random functions to be specified in the ini file. Similarly, lifetime is a volatile parameter used to schedule destruction events. A minimum guaranteed lifetime can be specified using the minLifetime parameter.

Node factory is not aware of the topology management component; its role is strictly limited to managing the node population. Generated node types must incorporate the `TopologyControlNIC` component, described in Section 3.3, which provides the ties between individual nodes and the graph topology.

3.2 Topology Management

`Topology` is a OMNeT++ compound module, consisting of a master controller, `TopologyControl`, and a *generator instance*. The controller manages the graph data structure and calls the generator upon creation and deletion of nodes as well as for periodic graph maintenance. Generator classes, which inherit from a base generator class, can be specified at run time in the ini file, providing an easily extendable and modular random graph generation mechanism, as further discussed in Section 4.

`Topology` works in conjunction with the `TopologyControlNIC` component, its per-node local counterpart, shown in Figure 1(b). The NIC calls the public method `registerNode` of `TopologyControl` upon its initialization. Similarly, the NIC calls `unregisterNode` when finalizing. `TopologyControl` handles these calls in a generic manner, simply handing the node off to the generator instance, which handles insertion and removal of vertices v in the dynamic graph in accordance with the implemented random graph algorithm.

`Topology` takes the following parameters:

topologyGenerator: The name of a topology generator class implementing the `IBasicGenerator` interface.
updateInterval: The interval in seconds between updates of the edge structure of the graph.
snapshotFile: The name of a file for storing snapshots of the graph topology for off-line analysis.

An example of a modelled communications network is shown schematically in Figure 2. A number of node instances, each a OMNeT++ compound module, map to a random graph $G^t = (V^t, E^t)$ in which vertices $v \in V^t$ represent the instantiated compound modules one-to-one, while

edges $e \in E^t$ represent their communications channels. The isomorphic representation of the network at time t is shown in Figure 2(b).

3.3 The Communications Interface

`TopologyControlNIC`, must be present in every compound module which is to be incorporated in the random graph. It calls registration and de-registration functions of the global topology management component as described previously.

NICs are made aware of changes in their local neighborhood in the graph topology by a signal message from the `TopologyControl` component. In response, NICs request a list of currently reachable neighbors and send a signal to the protocol, optionally with a delay to simulate failure detection latency.

Inter-node communications are implemented with direct message passing, using the OMNeT++ `sendDirect` method with the optional *propagationDelay* and *duration* parameters.

We choose the direct message passing mechanism over the channel models already present in OMNeT++, and the various libraries, primarily to allow the manipulation of a strictly virtual representation of the communications graph, internal to the topologyControl instance. This method also eliminates the overhead associated with maintaining a channel based network as well as keeping message complexity to an absolute minimum.

`TopologyControlNIC` takes the following configuration parameters:

dataRate: The data rate in Kbps.
bitErrorRate: A volatile parameter for the bit error rate.
processingDelay: The processing delay in seconds.
propagationDelay: The propagation delay in seconds.

A simple BER loss model is currently implemented in which messages are deleted (by the sender) upon a single bit error. The bitErrorRate, processingDelay and propagationDelay parameters are volatile, allowing any of the OMNeT++ random functions to be specified at run-time in the ini file. The transmission delay is computed from the packet bit size and data rate in the usual manner.

4. SIMULATING A RANDOM GRAPH

Management of edges in the random graph is handled by a *generator* instance – a plug-in companion to the `Topology-Manager` described in Section 3.2. The generators implement the interface `IBasicGenerator` and inherit from a base generator module, `BasicGenerator`. This architecture allows for modular implementation of diverse graph algorithms as well as run-time configuration of graph generation, simply by specifying a generator class name in the ini file.

`BasicGenerator` and derived components expose the following public methods:

void addNode: Adds a vertex to the graph data structure and creates edges in accordance with the implemented algorithm.

void removeNode: Removes a vertex from the graph. All incident edges are also removed.

bool update: Periodic updates of the graph edges to simulate dynamic effects other than node churn.

void constructInitialTopology: Called after initialization of the topology manager to create edges between nodes instantiated at time zero.

Random graph construction algorithms are implemented by inheriting from `BasicGenerator` and overriding its methods. Let us now consider two examples of such implementations.

4.1 The Binomial Graph

The binomial graph $g_{n,p}$, also known as a Erdös-Rényi graph [6], is defined classically as one of the set of possible graphs created by connecting each of the n vertices to any other vertex with probability p. This algorithm is the basis for the implementation of `GnpRandomGenerator`. A timeline for the evolution of a graph for $p = 0.2$ is shown in Figure 3.

We abuse the binomial algorithm slightly by applying it to a growing graph, as sketched in simplified form in Algorithm 1. A new node u is initially connected to one existing vertex selected uniformly at random. An expected number of links to pre-existing nodes is then created in accordance with the linking probability p. We implement the `constructInitialTopology` method of the generator to construct the initial graph of nodes added at time zero in one call to help preserve the characteristics of the $g_{n,p}$ algorithm.

Periodic graph maintenance is implemented as shown in Algorithm 2. A churn probability p_C is defined for the generator, specifying the probability of a single link being reassigned per time unit. We begin by iterating through E and removing any e with probability p_C per unit of time. Similarly, new links are added to any $v \in V$ with probability $p_C \cdot |E|/|V|$ per unit of time.

4.2 The Barabási-Albert algorithm

We implement the Barabási-Albert (BA) algorithm [3] for growing scale-free random graphs in the `BarbasiAlbertGenerator` module. The graph construction is similar to that described earlier, but the probability of a new node u forming a link to a peer u is $p(u) = k_u / \sum_{v \in V} k_v$, where k_v is the degree of a node $v \in V$. Under this model, new nodes tend to form associations with nodes of high degree, consistent with the preferential attachment model. The BA algorithm applies to a growing graph. Hence, in contrast to the `GnpRandomGenerator`, we implement edge creation at

---

Algorithm 1 Dynamic $g_{n,p}$ graph construction – addition of node u

{Upon registration of node u}
insert u into V, the vertices collection
if u is the only node **then**
   return
else if $|V| = 2$ **then**
   createLink(u,v)
else
   select node $v \in V$ uniformly at random.
   createLink(u,v)
   for $z \in V$ **do**
      createLink(u,z) with probability p
   end for
end if

Algorithm 2 Dynamic $g_{n,p}$ graph maintenance

{Executed periodically, T_Δ is the time units since last update.}
$p_r \leftarrow T_\Delta \cdot p_C$
$p_a \leftarrow T_\Delta \cdot p_C \cdot |E|/|V|$
for each $e \in E$ **do**
   remove e with probability p_r
end for
for each $v \in V$ **do**
   do with probability p_a: pick a neighbor $u \in V$ uniformly at random and add edge (v, u)
end for

---

time zero on a per-node basis in the `addNode` method; the `constructInitialTopology` call does nothing.

4.3 Graph Dynamism

Dynamic effects – node churn and link reassignments – have a side effect which we must consider. If a deleted node is a cut vertex in the graph, then we have partitioned the graph. Similarly, edge reassignments can easily lead to generation of multiple connected components, especially in sparsely connected graphs. Real-world networks, especially sparse ones, have generally no guarantees of connectivity so this behaviour may be acceptable. However, severe graph partitioning may lead to confusion regarding interpretation of results. Hence, we *optionally* guarantee that the graph is a single connected component despite node churn and link reassignments.

One method of guaranteeing connectivity despite churn is simply to connect the k neighbours of a deleted node u with $k-1$ links, thereby guaranteeing that all nodes belong to the same connected component in case u is a cut vertex. Deleting an edge may also partition the graph, but no simple method can be used to counter this without some knowledge of the graph structure.

Hence, we consider methods which allow us to determine, at least probabilistically, if a deleted node (edge) is a cut vertex (cut edge). Several algorithms can be considered, including union find, Karger's algorithm [14] for identifying min cuts or Westbrook and Tarjan's [24] on-line algorithm for identifying biconnected components.

We use the following alternative local search method: Upon deletion of node u, we randomly select a node $v \in \Gamma_u$, the neighbourhood of u. Let $\mathbf{l} = \Gamma_u \setminus v$ be a list of the previous
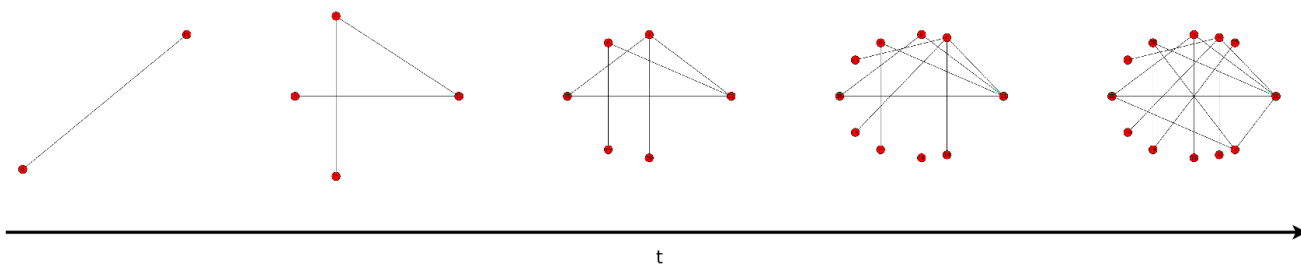
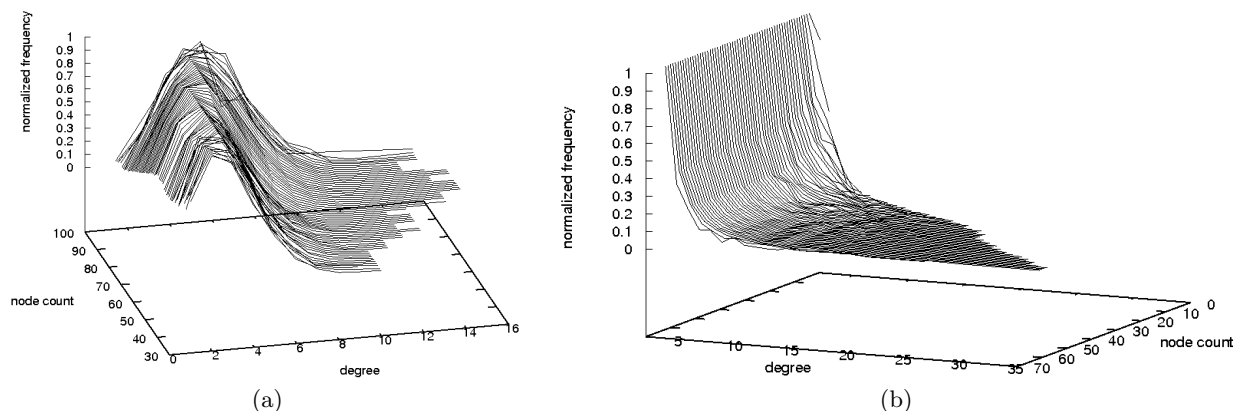Figure 3: $g_{n,p}$ graph growth. $p = 0.2$.

(a)

(b)

Figure 4: Degree distributions for growing random graphs, averaged over 50 runs. Arrivals follow a Poisson process with mean inter-arrival time of $1/\lambda = 60s$. Initial population is zero. (a) $g_{n,p}$ binomial random graph with $p = 0.05$. (b) Barabási-Albert random graph with $m_0 = 3$ and $m = 2$.

neighbours of u, excluding v. We now eliminate recursively from $\mathbf{l}$ all nodes which are reachable in $\leq k$ hops from v. If $\mathbf{l}$ is empty we stop and determine that the graph remains connected without intervention. However, if $\mathbf{l}$ is non-empty when the recursive search terminates, we create edges (a, b) between each $a \in \mathbf{l}$ and vertices b selected randomly from $\Gamma_u \setminus \mathbf{l}$. Similarly, if a removed edge (a, b) is determined to be a cut edge, we create a new edge (a', b') where a' and b' are selected from the Γ_k neighbourhood of a and b, respectively.

4.4 Validating the Random Graph Construction Algorithms

Let us now briefly and rather informally analyse the characteristics of the generators described.

Binomial graph generation. Figure 4(a) shows the degree distribution of a growing $g_{n,p}$ graph. Observe that the node degree may quite plausibly correspond to the Poisson distributed node degree expected for a classical binomial graph. However, more extensive statistical tests need to be performed in order to determine the characteristics of this particular graph generator. Hence, we do not claim any guarantees on the characteristics of the evolving graph with respect to those of the classical static $g_{n,p}$ graph at this time.

BA graph generation. The degree distribution for a growing Barabási-Albert graph is shown in Figure 4(b). Note the characteristic exponential degree distribution of the BA graph which gives it its scale-free properties. On a *log-log* plot, as shown in Figure 5, the degree distribution forms a straight line, as expected [2].
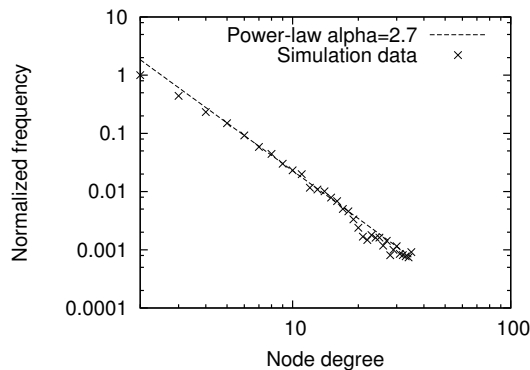

Figure 5: *log-log* plot of average node degree over all the data series shown in Figure 4(b).

5. SIMULATING A DISTRIBUTED AGGREGATION ALGORITHM

We now consider the modelling of a distributed algorithm using a dynamic random graph. We model the *Generic Aggregation Algorithm (GAP)* [4] – a tree-based distributed aggregation protocol designed for continuous network monitoring applications in dynamic networks. Building upon a self-stabilizing algorithm for distributed spanning-tree construction [5], GAP uses a dynamically maintained overlay to periodically send partial aggregates based on local state observations to the current parent. The parent in turn per-

forms *in-network aggregation*, that is, incorporates the partial aggregates received from its children into its current state by applying a local aggregation function and after a rate limitation delay produces a partial aggregate update of its own. This process continues until the updates reach the root of the tree, producing a fast and accurate aggregate approximation of the global system state. The protocol is efficient in terms of time and message complexity when compared to centralized monitoring alternatives. It also dynamically adapts to changes in topology at the rate at which neighbor discovery and failure detection is handled in the underlying networking layers.

We model the GAP protocol as a OMNeT++ simple module, implementing a generic `IProtocol` interface, which plugs into a networked node, as shown in Figure 1(b). Periodic updates of local inputs are generated by the sensor module. In our trials, the local input function is a monotonically increasing trend with random noise. The function also incorporates a sudden step every hour of simulation time to demonstrate the effects of transients on the aggregation algorithm.

In our implementation, the protocol is initiated by a querier which broadcasts a query message to the network. The propagation of the query initiates the tree construction, establishing the querier as the root. Nodes generated after protocol initiation receive a cached copy of the query from their neighbours as part of the initial post-discovery handshake process.

Failure detection, that is, the detection of unreachable neighbours, is assumed to lag behind the actual time of failure or departure, consistent with real-world neighbour discovery mechanisms such as periodic beacons. Hence, the aggregation topology will unavoidably lag behind the real topology image – the severity of this error depends on the frequency of updates. This is modelled by randomly delaying the notification of topology changes, discussed in Section 3.3, from NIC to protocol.

In our present implementation, unreachable nodes and their associated state are simply erased from the local neighbourhood table. This is consistent with the conservative caching strategy discussed by Dam and Stadler [4]. Alternatives include a cache-like strategy which may help to preserve partial aggregates associated with reappearing neighbours, for instance nodes which become reachable after brief failures.

The following experiments apply a dynamic random graph model to a generic networked measurement system. We can in this context think of router backbones in the Internet, whose relations have been shown to follow a power-law distribution [7]. Hence, we choose the Barabasi-Albert model [2] for the subsequent simulation experiments. Palmskog *et al.* describe aggregation in such network in the context of searches in the network of information [21]. Jonsson *et al.* consider the more general problem of top-k aggregation in a distributed network [13].

Let us now consider selected results from an experiment involving a number of networked nodes, as shown in Figure 1(b), in a Barabási-Albert random graph with $m_0 = 5$ and $m = 3$. Local (sensor) inputs are updated at random $\mathcal{N}(60s, 30s)$ intervals. The physical phenomenon observed or its units are not important in this experiment. Leaf nodes produce partial aggregate updates immediately upon receiving a local state update, whereas in-network nodes (aggre-gators) rate limit by delaying up to $240s$ before producing a update. However, aggregators produce a update immediately once all current children have delivered updates. The true aggregate is tracked by a *global observer* component which receives copies of all sensor update messages with zero delay.

Experiment I: Constant Population with Re-linking

Let us first consider the performance of the protocol under link re-assignments. 2500 nodes are generated at time zero and live for the duration of the simulation. The probability of link reassignment per link per minute of simulation time is given by the parameter p. Link re-assignments are initiated at $t = 2h$ and terminate at $t = 4h$ of simulation time; the topology is updated every $60s$ of simulation time. The topology manager notifies affected nodes (whose neighborhood changes) instantaneously by a control message to the associated NIC. However, the NICs delay notifications to the protocol by $1s + \mathcal{N}(0, 1s)$ to simulate a relatively long failure detection latency.

The global state is maintained by an `GlobalObserver` instance which receives zero-delay copies of all update messages sent in the GAP protocol.

Figure 6(a) shows the aggregate average (sum/count) computed by the distributed aggregation protocol and observed by the querier – the root of the spanning tree. Samples of the global aggregate are given with errorbars (95% confidence intervals) for comparison. Observe that the aggregate average computed by the distributed aggregation protocol is close to the global aggregate, although a slight tendency towards underestimation can be seen for the $p = 0.01$ graph.

Figure 6(b) shows the aggregate count of sensor observations, which can be interpreted as an aggregate node population count since each node contains exactly one observation at any given time. The count aggregate shows the effects of link reassignments on the protocol better than the average, which turns out to be relatively robust with regards to link failures. The count can be seen to fall sharply when link reassignments are initiated, but the protocol recovers quickly and maintains an reasonable and relatively stable estimate of node count for the duration of the simulation.

Experiment II: Node Churn

We now introduce node churn. Link reassignments are disabled, but the node lifetime parameter used to introduce dynamic node removal. Again, we use a Barabási-Albert graph with $m_0 = 5$, $m = 3$. 1500 nodes are generated at time zero and thereafter according to a Poisson distribution with $1/\lambda = 5s$. Node lifetime is randomly drawn from $\mathcal{N}(\mu, \sigma)$. However, lifetime for nodes generated at time zero is uniformly distributed $[0, \mu]$ to prevent the otherwise expected mass failures around $t = \mu$.

The churn rates chosen are relatively rapid, the lifetime being drawn from normal distributions with $\mu = \{4h, 6h, 8h\}$ and $\sigma = 4h$. This rate of churn is for example consistent with distributed cooperative sensing and processing networks in which individual contributors may appear sporadically.

Figure 7(a) shows the aggregate average computed by the distributed aggregation protocol and observed by the querier. Samples of the global aggregate are given with errorbars (95% confidence intervals) for comparison. Failure detection latency is $100ms + \mathcal{N}(0, 100ms)$. The results show that the performance of the aggregation protocol is quite ac-

ceptable, although a slight error can be seen for the more rapid churn rates.

Figure 7(b) shows the aggregate count for the previous dataset for each of the three cases, as observed by the querier. For comparison, the real number of live nodes, as logged by the factory instance, is indicated by markers. As expected, the aggregate count tracks the real count closely for the duration of the simulation. However, note the increasingly large dips as the churn rate increases. This is most likely due to the loss of highly connected "hub" nodes and the associated loss of measurement "mass".

Let us now increase the failure detection latency to $1s + \mathcal{N}(0, 1s)$ as in Experiment I. Now, observe that the protocol breaks down for the higher churn rates and begins to diverge from the actual node count. This example is precisely the motivation for thorough testing of dynamic distributed protocols under a variety of conditions.

6. CONCLUSIONS

We present a set of components for OMNeT++ which facilitate simulations using dynamically evolving random graph topologies. The components can be used for simulation of communications networks modeled as dynamically evolving graphs, but one may also apply them for simulations involving more abstract concepts, such as social networks. We outline the architecture of our set of components and their design. The plug-in based architecture for the graph generator modules is of particular interest, as it enables easy configuration and extendability of the system.

We outline a case study involving an implementation of the GAP distributed aggregation protocol in a dynamic random graph topology. The case study demonstrates the importance of thorough simulations of distributed protocols in a variety of graph topologies and under a range of conditions.

Our components can be extended in several ways. Firstly, we can implement a range of random graph algorithms in addition to the ones discussed here. Secondly, one may consider scripted generation and graph maintenance, similar to that proposed in the Opposim project [11], allowing for the use of external tools to generate dynamic scenario "scripts" beforehand.

7. ACKNOWLEDGEMENTS

8. REFERENCES

[1] F. Bai and A. Helmy. *A Survey of Mobility Models in Wireless Adhoc Networks*, chapter 1. 2008.

[2] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509 – 512, 1999.

[3] A.-L. Barabási and E. Bonabeau. Scale-free networks. *Scientific American*, 2003.

[4] M. Dam and R. Stadler. A generic protocol for network state aggregation. In *RVK 05*, Linköping, Sweden, June 2005.

[5] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic system assuming only read/write atomicity. In *Distributed Computing*, pages 3–16, 1993.

[6] P. Erdös and A. Rényi. On random graphs. *Publ. Math. Debrecen*, 6:290–297, 1959.

[7] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 251–262, New York, NY, USA, 1999. ACM.

[8] T. Gamer and M. Scharf. Realistic simulation environments for IP-based networks. In *OMNeT++ Workshop*, March 2008.

[9] Georgia tech internetwork topology models. [online] http://http://www.cc.gatech.edu/projects/gtitm.

[10] N. Gershenfeld, R. Krikorian, and D. Cohen. The Internet-of-Things. *Scientific American*, October 2004.

[11] O. R. Helgason and K. V. Jónsson. Opportunistic networking in OMNeT++. In *OMNeT++ Workshop*, 2008.

[12] M. Jelasity, A. Montresor, and O. Babaoglu. T-Man: gossip-based fast overlay topology construction. *Computer Netw.*, 53(13):2321–2339, 2009.

[13] K. V. Jónsson, K. Palmskog, and Ý. Vigfússon. Secure distributed top-k aggregation. In *ICC*, 2012. accepted.

[14] D. R. Karger and C. Stein. A new approach to the minimum cut problem. *J. ACM*, 43:601–640, July 1996.

[15] J. Leskovec and E. Horvitz. Planetary-scale views on an instant-messaging network, 2008.

[16] F. Li and Y. Wang. Routing in vehicular ad hoc networks: A survey. *IEEE Vehicular Tech. Mag.*, 2(2):12 –22, june 2007.

[17] M. May, G. Karlsson, Ó. Helgason, and V. Lenders. A system architecture for delay-tolerant content distribution. In *WreCom*, Rome, Italy, October 2007.

[18] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: An approach to universal topology generation. In *MASCOTS*, Cincinnati, Ohio, USA, Aug 2001.

[19] S. Milgram. The small-world problem. *Psycology Today*, 1(61), 1967.

[20] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *P2P*, Seattle, WA, USA, September 2009.

[21] K. Palmskog, A. Gonzalez Prieto, C. Meirosu, R. Stadler, and M. Dam. Scalable metadata-directed search in a network of information. In *Future Network & Mobile Summit*, 2010.

[22] A. Varga. The OMNeT++ discrete event simulation system. In *ESM*, June 2001.

[23] D. Watts and S. Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393(6684), 1998.

[24] J. Westbrook and R. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.

[25] W. Willinger, D. Alderson, and J. C. Doyle. Mathematics and the internet: A source of enormous confusion and great potential. *Notices of the AMS*, 56(5), 2009.

[26] J. Winick and S. Jamin. Inet-3.0: Internet topology generator. Technical Report UM-CSE-TR-456-02, University of Michigan.

[27] J. Yick, B. Mukherjee, and D. Ghosal. Wireless sensor network survey. *Computer Networks*, 52(12):2292 – 2330, 2008.
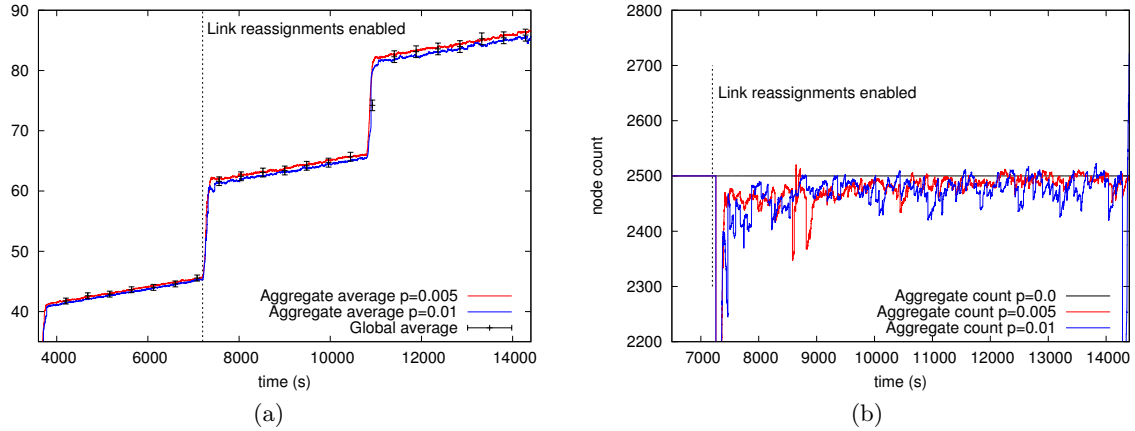
Figure 6: GAP executed in a BA graph ($m_0 = 5$, $m = 3$). 2500 nodes are generated at time zero and live for the duration of the simulation. p is the probability of link reassignment per link per minute of simulation time. (a): Aggregate average observed by root. Error bars show the globally observed aggregate. (b): Aggregate count observed by root.
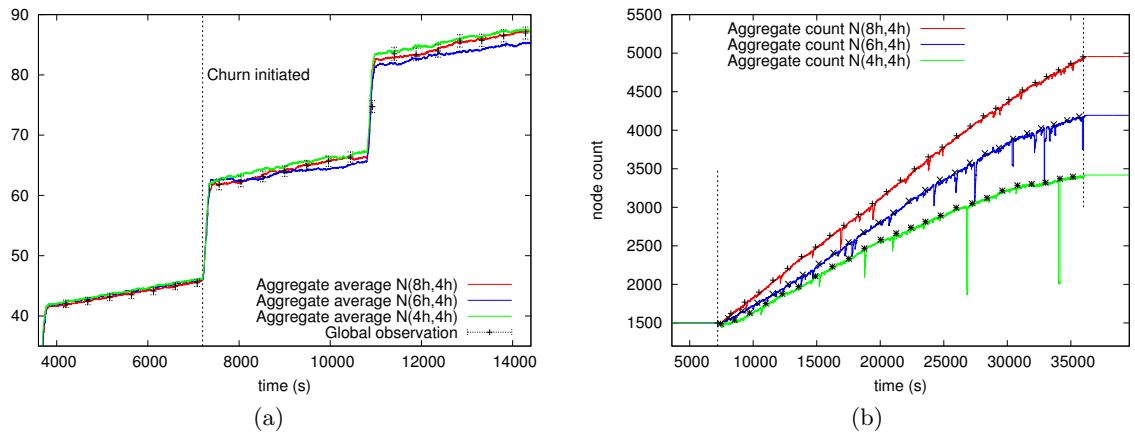

Figure 7: GAP executed in a BA graph ($m_0 = 5$, $m = 3$). 1500 nodes are generated at time zero and thereafter according to a Poisson distribution with $1/\lambda = 5s$. Node lifetime is randomly drawn from $\mathcal{N}(\mu, \sigma)$; lifetime for nodes generated at time zero is uniformly distributed $[0, \mu]$. Failure detection latency is $100ms + \mathcal{N}(0, 100ms)$. (a): Average observed by root. Error bars show the globally observed aggregate. (b): Count observed by root. True live node count shown by markers.
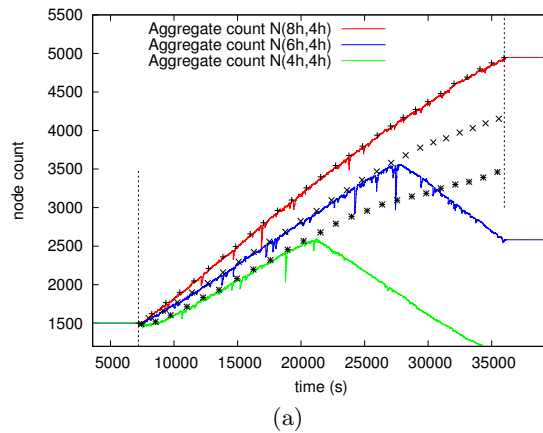

Figure 8: GAP executed in a BA graph ($m_0 = 5$, $m = 3$). Same parameters as for the case shown in Figure 7(b), except failure detection latency is $1s + \mathcal{N}(0, 1s)$.