Opportunistic Network Environment simulator

Ari Keränen
ari.keranen@tkk.fi

Helsinki University of Technology
Department of Communications and Networking
Special assignment

Supervisor: Professor Jörg Ott

May 29, 2008

Contents

1 Introduction **1**

2 Background **3**

3 The Opportunistic Network Environment simulator **6**

3.1 Mobility modeling 7

3.2 Routing simulation 9

3.3 External event and reporting frameworks 11

3.4 Running simulations 12

3.4.1 Graphical User Interface mode 12

3.4.2 Batch mode 14

3.4.3 Post-processing 15

3.4.4 Configuring scenarios 16

3.5 Software architecture 17

3.5.1 Movement models 18

3.5.2 Routing modules 19

3.5.3 Extending the simulator 21

3.6 Limitations 23

3.7 Results 24

4 Conclusions **25**

A The ONE ReadMe **I**

Acronyms

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
DTN	Delay Tolerant Networking
GIS	Geographic Information System
GPL	GNU General Public License
GPS	Global Positioning System
GUI	Graphical User Interface
MAC	Media Access Control
MANET	Mobile Ad-hoc NETwork
ONE	Opportunistic Network Environment simulator
POI	Point Of Interest
RAM	Random Access Memory
RWP	Random WayPoint
TTL	Time To Live
WKT	Well Known Text
WLAN	Wireless Local Area Network

Chapter 1

Introduction

Today many personal mobile devices include capabilities to communicate with infrastructure networks but also with each other. The latter can be used to form ad-hoc networks where common infrastructure is no longer needed for communication among hosts participating in the network. Ad-hoc networks can also help mobile nodes to reach infrastructure if some node in the network is able and willing to act as a gateway and possibly other nodes as relays for the traffic. Networks can be formed this way as long as the node density is large enough so that there exists possible end-to-end paths between all nodes wanting to communicate. However, if the node density decreases or the connectivity breaks for some other reason (e.g., the radios are switched off occasionally), traditional network communication protocols are no longer able to provide means for multi-hop communication.

Delay Tolerant Networking (DTN) [10, 4] is a communication networking paradigm that enables communication in environments where there may be no end-to-end paths, communication opportunities come and go and their interval can be very long and not even known beforehand. Routing messages in this kind of environments can be quite different compared to traditional networks. This has created a need to find new routing protocols that take efficiently into account the distinct nature of these networks. Different approaches can be tested and evaluated by simulation, but the simulation results are really useful only if they are a result of somewhat credible simulation scenarios.

The goal of this study is to add more realism to the simulations of Delay Tolerant Networks. For this purpose we created a new simulation environment called Opportunistic Network Environment simulator (ONE). Unlike other DTN simulators, which usually focus only on routing simulation, the ONE combines mobility modeling, DTN routing and visualization in one package that is easily extensible and provides a rich set of reporting and analyzing modules.

Chapter 2

Background

One hindrance for research on DTNs is the lack of good simulators. Many simulators exist for MANETs (e.g., ns2 [19] and OMNeT++ [27]) and also for DTN routing (e.g., dtnsim [13] and dtnsim2 [20]) but the former lack good DTN support and the latter concentrate solely on routing simulation. The DTN routing simulators have also seen little evolution since their first public release and implementations of recent routing protocols don't exist or are not publicly available.

Another problem for a simulator that considers only routing is that it needs input data that tells the routing protocols when a network link between two DTN nodes is up and when it is down. This data can be generated based on some random process (e.g., just drawing contact durations and endpoints from a pseudo random number generator), it can be derived from real-world trace (e.g., using CRAWDAD [5] data) or it can be derived from mobility simulation. The problem with the first approach is that values from some random distribution are hard to prove to have some validity behind them. For example, even if human interactions may seem somewhat random, humans usually do have some purpose for their actions and for meeting certain people and visiting some places more often than others. Capturing this in a simple, or even complex, random number distribution would be challenging, to say the least.

The real-world traces obviously capture real human behavior well but they have other kinds of problems. The number of usable traces has been

low even though the CRAWDAD project is bringing some relief into this. Unfortunately, the existing traces have low spatial and temporal granularity. To save battery life of the mobile devices that are used to track people, the interval the devices scan for others has been kept low [7]. This results in missing possible contacts but also in that we don't know how long the actual contacts lasted — we just know how many scanning intervals they were in contact with each other. This information can be of course used for many simulations, but if we are interested in utilizing even the shortest contacts, or want to experiment with different scanning intervals, these traces are not that useful anymore. If we had traces of the nodes' exact locations, we could derive contact times from that information by setting a simulated link between two nodes up when they are within certain distance (the radio range of the simulated device) of each other. Some of the wireless traces that are available support this to some extent because they have logs of the people who have been within the radius of some WLAN base stations whose location is known. From that information the nodes' approximate location can be estimated but the spatial granularity is not high enough for simulating devices with small or moderate radio range (e.g., Bluetooth with 10 meter radius). Yet another problem with the gathered traces is that the population in them is fixed and often highly specialized. One cannot increase the number of people that participated in the trace after the trace is gathered and the number is often quite low because of practical reasons. Also, so far the traces consist of more or less selected group of people (e.g., people participating in a conference or working in a research laboratory) and their behavior is not likely to be representative of some other group of people in other kind of situation. For these reasons, the real-world traces do not provide comprehensive solution for the connectivity data problem.

A third approach is to simulate the movement of the nodes and derive contact information from that. With this approach the temporal and spatial granularity can be set as high as what is needed for good simulation. Also, the number of nodes and their behavior is easily varied for different scenarios and sensitivity analysis. Even with simple mobility model the connectivity pattern is easily more realistic than with some random distribution. Then again, finding a good and somewhat realistic mobility model is a harder task.

One of the simplest ones that is commonly used is the Random Waypoint (RWP) [14]. In RWP random coordinates in the simulation area are given to the node. Node moves directly to the given destination at constant speed, pauses for a while, and then gets a new, random, destination. This continues throughout the simulations and nodes using RWP move along these zig-zag paths. This creates simple, but quite artificial movement: hardly ever humans can move unconstrained using direct paths or select their destinations randomly in a square area.

Even if we had a good algorithm for a movement model, it would be hard to program and validate it without seeing how the nodes move if the model is used. The same goes for DTN routing simulation: even though the simulation progress can usually be followed from text output to the terminal console, it requires a considerable amount of expertise to be able to see and understand what is happening there. For this purpose a Graphical User Interface (GUI) is an efficient tool. For ns-2 simulations, the iNSpect [16] addresses the visualization problem, but for the DTN routing simulators there have not been similar tools available.

For these reasons, there appears to be a need for a simulator that has reasonable mobility modeling capabilities, integrated support for DTN routing, and ways for visualizing the simulation progress and results in an intuitive way. The next chapter presents our response to this need: the Opportunistic Network Environment simulator.

Chapter 3

The Opportunistic Network Environment simulator

To make complex DTN simulations more feasible and understandable, we created a new simulation environment that combines movement modeling, routing simulation, visualization and reporting in one program. Movement modeling can be done either on-demand using the integrated movement models or movement data can be imported from an external source. The node position data that the movement models provide is then used for determining if two nodes can communicate and exchange messages. This information can be exported for routing simulation in external simulators (such as dtnsim) or it can be given to the internal routing modules which implement several different DTN routing algorithms. The internal routing modules perform operations on the messages on their own, but they can also be commanded using event generator modules or external traces. The movement modeling and routing simulation is interactively observable in the simulator's GUI and report modules can gather data of the simulation for further analysis or interaction with other programs. An overview of the whole process is depicted in figure 3.1.

The core of the ONE is an agent-based discrete event simulator. To make it suitable and efficient enough for simultaneous movement and routing simulation, it uses time slicing approach [2], so the simulation time is advanced in fixed time steps. The time slicing can be complemented by scheduling

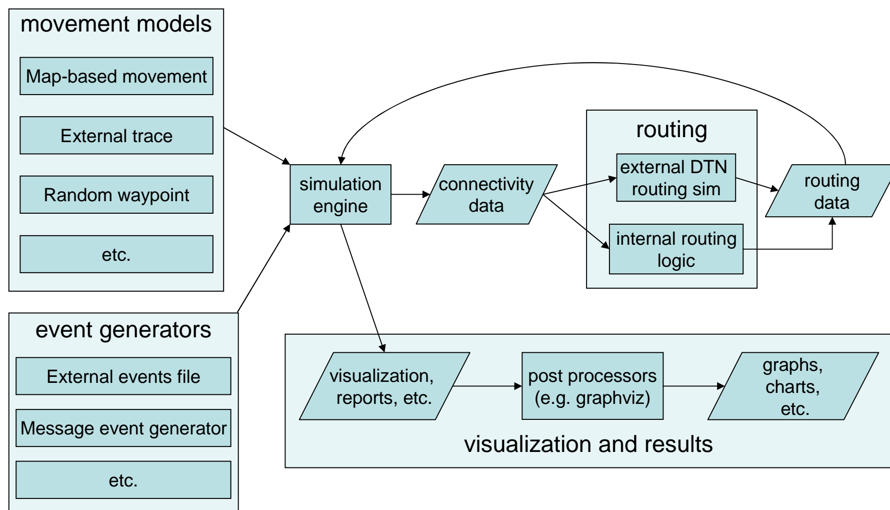

Figure 3.1: Overview of the ONE

update requests between the fixed time steps for higher simulation time resolution.

The simulations can contain any number of different types of agents, i.e., wireless nodes. The nodes are grouped in *node groups* and a one group shares a set of common parameters such as message buffer size, radio range and mobility model. Since different groups can have different configurations, creating e.g., a simulation with pedestrians, cars and public transportation is possible. All movement models, report modules, routing algorithms and event generators are dynamically loaded into the simulator so extending and configuring the simulator with different type of plugins is made easy for users and developers: just creating a new class and defining its name in the configuration file is usually enough.

3.1 Mobility modeling

Mobility models dictate how the nodes move during the simulation. Three different types of mobility models were initially implemented for ONE. For reference purposes, even despite of its shortcomings, ONE includes the basic Random Waypoint movement model. For more realistic mobility scenarios

ONE provides variety of map-based movement models which constrain the node movement to predetermined paths. Finally, ONE also supports importing of mobility data from external sources.

Map-based movement models accept map data that is described using a subset of the Well Known Text (WKT) format. WKT is an ASCII based format that is commonly used in Geographic Information System (GIS) programs. Since basically all the digital map data is available in some format that GIS programs understand, they are usually relatively easily convertable to the form supported by the ONE. Also, GIS programs can be used as powerful map editors for the ONE. The free, Java-based, open source GIS program OpenJUMP [21] was used for editing and converting the maps in our experiments. In map-based models, the nodes move using only the roads and walkways of the map area. Different node groups can be configured to use only certain parts of the maps which can prevent, e.g., cars driving on pedestrian paths or inside buildings (if paths in buildings are also defined).

The simplest Map-Based Model, *MBM*, places nodes randomly in the map area and moves them forward on the path segments until they hit the end of the road and turn back or end up in an intersection. In intersections, nodes using MBM select randomly a new direction but do not head back where they came from. When a node has traveled a configurable distance, it stops for a (configurable) while, and then continues its journey. The more advanced version of MBM, Shortest Path Map-Based movement Model, *SPMBM*, also initially places the nodes in random places but selects a certain destination in the map for all nodes and uses Dijkstra's shortest path algorithm [6] to find the shortest path to the destination. When the destination is reached, the node waits for a while and selects a new destination. Normally all the places in the map have equal probability of being chosen as the next destination, but the map data can also contain Points of Interest (POIs). POIs are grouped in POI groups and every node group can have configurable probability for choosing a POI in certain group as the next destination. POIs are useful for modeling e.g., restaurants, tourist attractions and other places where people tend to gather. Some nodes can also have predetermined routes that they travel on the map. This kind of Route-Based Models, *RBMs*, are good for simulating e.g., bus and tram

routes. Routes consist of map points that model the stops on the route and nodes wait on every stop for some time before continuing, using the shortest path, to the next stop. Both POIs and routes can be defined using the same GIS programs that are used for converting the map data.

The external mobility model takes a set of timestamped coordinates as the input and moves the simulated nodes accordingly. This model can be used e.g., with GPS traces of real users or synthetic mobility traces generated by other programs. One program that is suitable for creating synthetic traces is the Transportation Analysis and Simulation System (TRANSIMS) [18]. TRANSIMS contains an agent-based microsimulator that takes into account activities of individuals and their interactions when they move in simulated metropolitan area and use different transportation options. The agent movement data can be exported from TRANSIMS and imported into ONE's external mobility model and used for DTN simulations.

3.2 Routing simulation

While the mobility models decide where the nodes should move next, the routing modules get to decide where the messages, or bundles, end up. The ONE has six implementations of different well known routing algorithms and also a passive routing module that can be used for interaction with external DTN routing simulators.

The active routing modules included in the ONE are: *First Contact* [13], *Direct Delivery* [24], *Spray and Wait* [25] (normal and binary), *Epidemic* [26], *PRoPHET* [17] and *MaxProp* [3]. When two (or more) nodes meet and there is a chance to exchange messages, all of the routing modules first check if they have any messages that are destined for the other node and try to send them. If the message was already earlier received by the node, it declines receiving it and other messages can be tried. After all, if any, such messages are exchanged, the behavior with rest of the messages depends on the routing algorithm.

The most simple routing modules are the Direct Delivery, Epidemic and First Contact. The Direct Delivery module does not start any further transactions after exchanging the deliverable messages since it will send messages

only if it is in contact with the final recipient. This saves buffer space and bandwidth but is not obviously an optimal approach in many cases if a high message delivery probability is the goal. The Epidemic routing module uses quite different approach since after two nodes have exchanged the deliverable messages, it tries to exchange also all the other messages until both nodes have the same set of messages or the connection breaks. If we had unlimited buffer space and bandwidth, this would result in maximal spreading of the messages throughout the nodes and therefore also to maximal delivery probabilities. However, if the buffer space and/or bandwidth is limited, Epidemic routing is also likely to waste a lot of resources. For example, a message that was delivered shortly after it was created could be transmitted and stored by all the nodes for a long time even if there is no longer use for it. This way the message is using resources that could be better used for the messages that are not yet delivered. While also the First Contact module sends as many messages to the other node as it has time, it removes the local copy of the message after a successful transfer. This results in only a single copy of every message in the network. To prevent two nodes who stay in contact for a long time exchanging the same messages back and forth, the receiving node accepts a message only if the message has not passed through it before. Unfortunately, there are no guarantees that the first node that is met is a better candidate than the previous node carrying the message, so First Contact is not likely to achieve very high delivery probabilities either.

The Spray and Wait works a bit like the Epidemic but is a bit more complex routing module since it restricts the amount of copies that are spread in the network. This is done by letting each created message to replicate only a certain amount of times. A node that has more than one copy of the message left, can give either a one copy to another node (the *normal* mode) or half of the copies (the *binary* mode). If the node has only a single copy of the message left, it is transmitted only to the final recipient. By using a different amount of initial copies, Spray and Wait can balance between high diffusion of messages and excess use of resources. PRoPHET and MaxProp take the complexity a bit further since they keep track of which node has been in contact with which nodes. This information can be used to reason if a certain node is a good candidate for delivering a message

to another node by assuming that if two nodes have met before, they are more likely to meet (soon) again. While PRoPHET checks if another node is more likely to meet the final recipient, MaxProp takes this idea further and uses Dijkstra's algorithm to calculate whole paths from node to node using the meeting probabilities. MaxProp also uses acknowledgments of delivered messages that help flushing redundant messages from the network.

The passive routing module can be used as an interface to other DTN routing simulators. If a DTN routing simulator can output timestamped information about message related events (creating, relaying and removing messages), this data can be input to ONE for analysis and visualization. For example, dtnsim2's debug trace can be converted to suitable form for the ONE. If the contact schedule for dtnsim2's input is created using the ONE, the message routing and mobility modeling can be visualized and inspected in the GUI in similar way as if the routing was done by the ONE.

3.3 External event and reporting frameworks

While the routing modules could spawn new messages whenever needed, it is often more convenient to have a routing module independent way of creating them. Also, for interaction with other programs, there needs to be a way to import message and connectivity related events to the simulation. The external events framework is the solution for this.

There are two different ways to import events to the ONE: trace files and event generator modules. A trace file is a simple text file that has timestamped events such as creating a message, removing a message from the message buffer or setting up a new connection. The trace files can also be saved and loaded in binary mode to save time from parsing text files. Event generator modules are normal Java classes that can dynamically create the same events as the trace files. While the event traces are usually generated with a script or converted from some other program's output, the event modules can be configured using the same settings system as the rest of the simulator. The ONE supports multiple simultaneous event generators and their events are automatically interleaved in the simulation.

While the external events provide input for the simulation, the most

important output is generated by the report modules. Report modules can register to connection, message, and/or movement related events. Whenever such an event happens in the simulation, the related method is called for all registered report modules and a reference to the relevant objects (e.g., nodes and messages) is given to the report module. The report modules usually either write information about the event to a report output file if it was relevant or just store the information to an internal data structure for creating a summary when the simulation is done.

The ONE has already numerous reporting modules, for example, for message statistics (such as delivery probability and round-trip times), node contact and inter-contact times and message delivery delays and distances. Other interesting report modules are the ones used for communication with other programs. The node connectivity information can be directly reported in a suitable form for dtnsim2 simulations so the ONE can be used as a mobility simulator for it. Also, another report module is able to output mobility traces suitable for ns2's Monarch [23] mobile node extension.

3.4 Running simulations

The ONE can be run in two different modes: batch and GUI. The GUI mode is especially useful for testing, debugging and demonstration purposes and the batch mode can be used for running large amount of simulations with different set of parameters. Both modes can include any number of report modules which produce statistics of the simulation. These statistics can be further analyzed with post-processing tools to create different kind of summaries, graphs and plots.

3.4.1 Graphical User Interface mode

In the GUI mode the simulation is visualized in real time as shown in the figure 3.2. The largest part of the GUI is taken by the play field view which contains a bird's-eye view of the geographical simulation area. Node locations, their radio range, current paths, amount of messages etc. are visualized on the play field view. If the current movement model is map-based, also the map path segments are drawn in the view. Additionally, a

background image, such as an aerial photograph or a raster map, can be displayed under the other graphics. The play field view can be centered by clicking with a mouse button and zoomed with the mouse wheel.

The simulation speed can be adjusted with the controls in the upper part of the GUI and any node can be selected for inspection with the node selector panel on the right side. When a node is selected for inspection, its current location and amount of messages is displayed. Any of the messages the node is carrying can be taken for further inspection and also routing module specific information can be retrieved. The GUI also keeps track of notable events and displays them in the event log panel. By clicking a node or message name in the log panel, more information will be shown about that node or message. The event log controls panel can be used to adjust which events are shown in the log and the simulation can also be made automatically pause in case of some type of event.

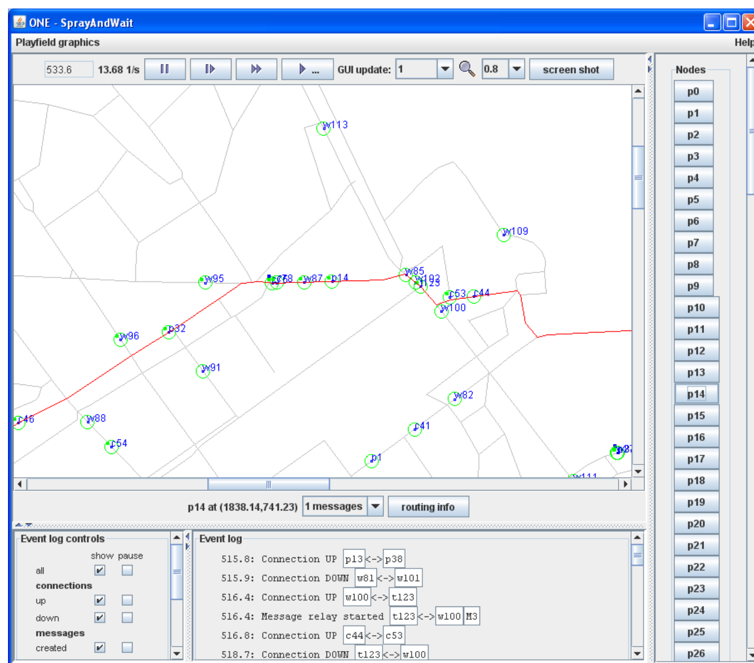

Figure 3.2: ONE screen shot (map data copyright: Maanmittauslaitos, 2007)

When a new movement model or routing module is created, the GUI can give an intuitive view on how it works and whether the operations

make much sense. Since node locations are shown on the map, by following their movement for a while, the movement model's creator can see if the nodes move like they were supposed to move and end up in places where they were expected to go. This also applies to the routing modules: the simulation can be automatically paused when, e.g., a message is transferred between two nodes. Then, the routing module's state can be inspected to see if the message was really supposed to be transferred or not. During the development of the ONE, these methods were extensively used to test and debug the simulator.

Also, when deciding on the simulation parameters, it may be sometimes hard to come up with reasonable values: for example, is 1000 nodes with a 50 meter radio range a lot in a 10 square kilometer urban area? Trying to deduce this analytically can be a tedious job, but running such a simulation with the GUI gives immediately an intuitive overview, e.g., on how well connected the nodes are and how fast the messages are spread in the network.

Finally, the GUI can be useful for demonstration purposes. Seeing nodes moving and messages being passed between them is often more intuitive than a text trace in a console window or seeing just the final result in a plot when the simulation is over.

3.4.2 Batch mode

When the movement models and routing modules seem to work as expected, and if there is no need for real-time visualization, it is more efficient to run simulations in the batch mode. The batch mode does not have a GUI so all the processing power can be used for the simulation.

In the beginning of every scenario the name of the scenario and the amount of scenarios left in this batch is printed to the console. If a scenario takes more than one minute to run, a progress message is printed once every minute that shows the current simulation time and the speed of the simulation (simulated seconds per second). All simulation results are gathered using the output of report modules.

For scenario configuration, the batch mode has a useful feature called *run indexing*. This allows easily defining a set of scenarios with varying set

of parameters and trying out different parameter combinations with only one configuration file. The idea in run indexing is that the user provides for some configuration parameters, instead of a single value, an array of values. The user also defines, with a command line parameter, how many simulation runs should be performed. When started, the simulator first selects the first value on each settings array and runs a simulation using those values. When the simulation is finished, the second value from each array is chosen for the next simulation and so forth. If an array's length is smaller than the amount of runs, the indexing wraps around, so the first parameter is selected after the last one. Therefore, if the amount of arrays and their length is chosen so that all array lengths have one as their smallest common nominator (e.g., by using prime numbers), all setting combinations can be run with a single configuration file. For example, three setting arrays with lengths of 3, 5 and 7 will result in $3 \times 5 \times 7 = 105$ different combinations.

3.4.3 Post-processing

When a set of report files have been created, either with GUI or batch mode, they can be further post-processed with suitable programs.

Two of the ONE's ready-made report modules produce files that are directly suitable for Graphviz [1] input. The *adjacency graphviz report* module creates node adjacency graphs such as shown in the figure 3.3. In the graph the nodes that have been in contact more than once have an edge drawn between them and nodes that have been more often in contact are drawn closer to each other. For example, it can be seen from the graph that tram nodes (marked with t), that use the same route, meet quite often during the simulation. Another Graphviz suitable module is the *message graphviz report* module which produces directed graphs of the delivered messages' paths. An example of a message graph, with all the messages that were sent from the node p1 to the node p2 during a simulation, is shown in the figure 3.4.

The ONE includes also a set of post-processing scripts that can automatically create, for example, bar graphs of message delivery probabilities or plot cumulative distribution of inter-contact times using gnuplot [12].

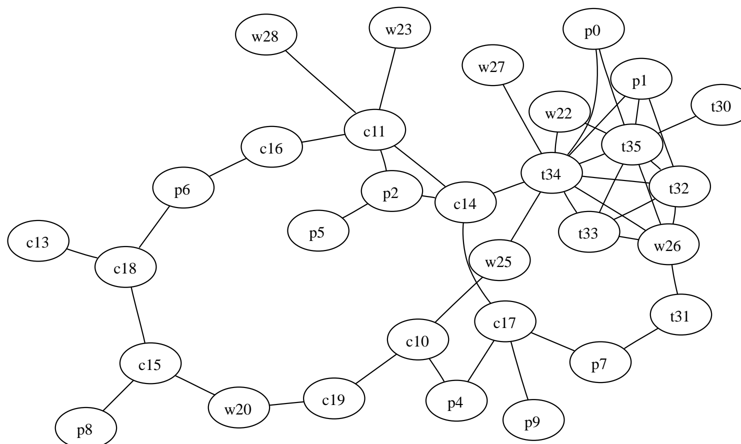

Figure 3.3: Example of a node adjacency graph

3.4.4 Configuring scenarios

For defining the simulation scenarios, a way to configure the simulator is needed. The ONE is configured using settings files which provide key-value setting pairs. The setting values can be strings, numeric values, or class names. Settings are grouped into name spaces so that same setting key can have different values in different contexts. Any settings object can have a primary and secondary name space: if the value that is requested is not found from the primary name space, it is looked up from the secondary name space. This way, a single entry is enough for defining some value for a whole group of settings (e.g., for all node groups) but still some or all groups can override these values by defining a new value in their primary name space.

Class name values are used for dynamically loaded components and a class with the specified name is looked from the related package. If such a class is found, the settings framework instantiates it giving the new object a reference to the settings object that is initialized to its name space.

String values are passed to the requesting object as such but the numeric values, integers and decimals, can contain kilo (k), mega (M) or giga (G) suffix for easily presenting large values which the settings framework automatically parses on behalf of the requester. Any key can also have more than

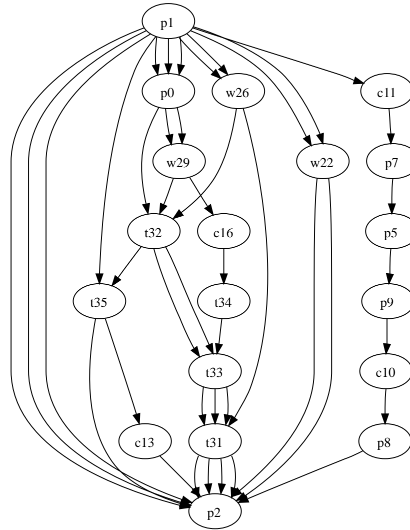

Figure 3.4: Example of a message graph

one value in which case the values are separated by commas. For example, value ranges and area sizes are presented this way. Further description and examples on how to configure the simulation environment are shown in the appendix A.

3.5 Software architecture

The simulator part of the ONE is written using the Java programming language. Different parts of the program are divided in different packages as described in the figure 3.5. The figure also depicts, roughly, the dependencies between the packages. Core components of the simulator, such as classes presenting a DTN host or a connection, are in the *core* package. GUI-related classes are in the *gui* package, which also has the *playfield* sub-package containing classes presenting graphical objects in the play field view. Generic user interface class and the text-based console class are stored in the *ui* package. The (G)UI classes instantiate *SimScenario* and *World* classes from the core package which in turn create instances of routing modules from the *routing* package and movement models from the *movement* package. During the simulation, routing and movement modules provide output for report

modules from the *report* package. The *test* package is not directly related to the simulator, but it contains a set of unit and system tests that can be run to check if the system performs as expected.

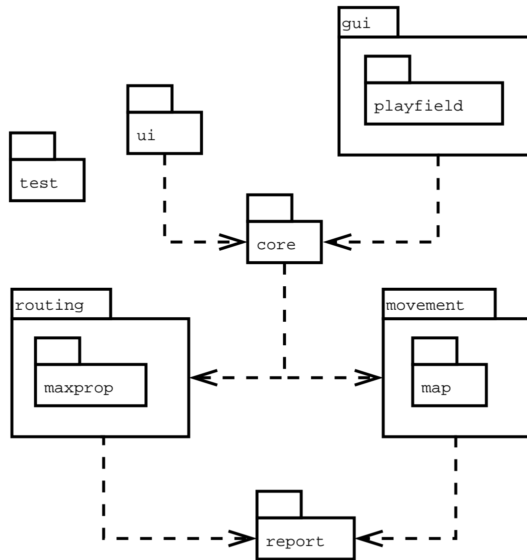

Figure 3.5: The ONE software packages

3.5.1 Movement models

Some of the most important classes in the movement package are shown in the figure 3.6. All movement models extend the *MovementModel* class which provides the interface for requesting a new path for a node and asking when the next path is available. The subclasses provide different implementations for these and implement this way different modes of movement. For example, *RandomWaypoint* objects gives out simple zig-zag path whereas *MapBasedMovement* objects restrict the path components to path segments defined in a map data. The *map* sub-package has utility classes for the map based movement models. The *SimMap* class provides a presentation of the map data and the *DijkstraPathFinder* class can use that data to find the shortest path between two map nodes. The *PointsOfInterest* class in turn takes care of reading POI data and selecting appropriate POIs according to the defined configuration.

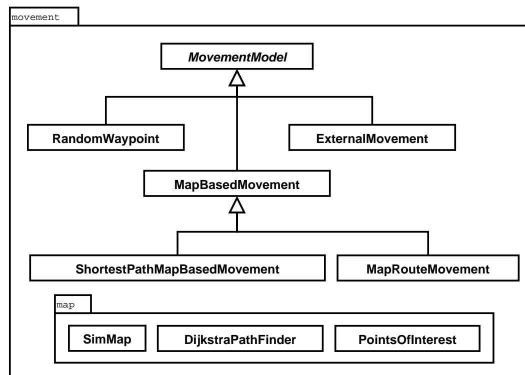

Figure 3.6: The movement package

The *MapBasedMovement* class extends the default movement model class by adding map-related features such as reading and caching map data. *MapRouteMovement* and *ShortestPathMapBasedMovement* movement models also use the map data and the DijkstraPathFinder class to move using shortest paths between route stops and and other map destinations.

3.5.2 Routing modules

The routing package, shown in figure 3.7, has a similar structure as the movement package discussed in section 3.5.1. All routing modules must extend the *MessageRouter* class which provides the basic interface and functionality for routing modules. The *ActiveRouter* class provides many utility functions that (currently) all active routing modules use. The *PassiveRouter* class does not need those, so it is inherited directly from the MessageRouter. The *MaxProp* routing module requires some helper classes which are stored in the *maxprop* package.

The MessageRouter class takes care of storing information of the messages the node currently has in its buffer, the messages it is currently receiving on its active connections, and about the messages that it has received as the final recipient and has moved to the application layer so they are no longer in the message buffer. When a node wishes to try to transfer a message to another node, it asks the related connection object to start the transfer, which in turn forwards the request to the other node. The other

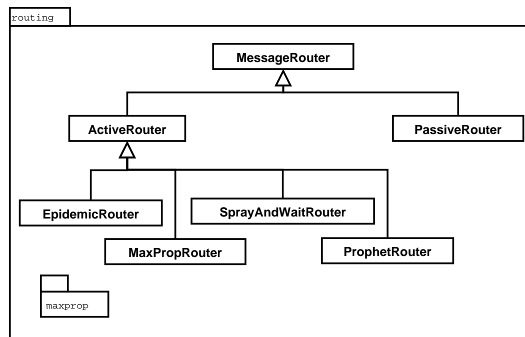

Figure 3.7: The routing package

node calls its router module's *receiveMessage* method and then the router module can check whether it wants to receive the message. The router module may reject the message, e.g., if it already has it in its message buffer. The MessageRouter class accepts any incoming messages, but the modules extending it may, and most likely will, have additional logic for accepting or denying messages. With the method return value the router module can also signal whether the other node should try to send the same message later again or just stop trying.

In addition to the message receiving method, the MessageRouter class also has methods that are called when a message was transferred successfully, when a transfer was aborted, when a new message should be created or when a message should be deleted from the buffer. The implementations for these methods are fairly straightforward and their most important task is to inform registered event listeners, such as report modules or the GUI event panel, about the events. The class also has two important methods that it does not provide implementations for: the *changedConnection* and *update* methods. The former is called every time a new connection comes up or an old connection goes down, and the latter is called on every update round. Active router subclasses have to provide implementations for both of these, since with them they are able to react to connection opportunities and also handle the ongoing transactions.

The ActiveRouter class extends the MessageRouter and provides implementations for the most common tasks, such as sending messages that can

be delivered to a directly connected host or trying to send a set of messages in certain order using a given set of connections. It also provides a simple implementation for the update method. The update implementation finishes ready transfers, aborts transfers if the connection was torn down before the message transfer finished and also drops messages whose time-to-live (TTL) has expired. A simple routing module does not need much more to function and the simplest modules extending the ActiveRouter have only a few lines of code.

3.5.3 Extending the simulator

As mentioned before, the ONE is meant to be an easily extendable simulation environment. Since the program is released under an open-source GPL license [11], anyone is free to modify the behavior of the simulator practically in any way they seem fit. However, to make things easier, we have introduced several extension hooks which can be used without need for any changes in the original source code. This allows sharing modules as plugins and using them in different versions without needing to patch the other parts of the simulator code. Routing modules, movement models, event generators and report modules are all dynamically loaded when the simulator is started using the Java reflection API. Due to the use of dynamic loading, user only needs to create a new class, define its name in the configuration file, and the simulator automatically loads it when the scenario is started. All these modules can also have any number of settings defined in the configuration files and these settings are accessible to the module when its loaded.

As described in the section 3.5, all routing modules must extend the *MessageRouter* class, but for implementing a routing algorithm, its usually more convenient to extend the *ActiveRouter* class. The most important one to override is the update-method. Finishing the ongoing transfers and also exchanging the deliverable messages can be delegated to the ActiveRouter superclass but further routing logic must be implemented by the routing module itself in the update method. Also, if a router module needs to keep track of all connections, even when its transferring messages, it should

override the changedConnection method for doing the necessary actions. For example, the PRoPHET routing module keeps track of nodes who have met each other this way.

New movement models can be created by extending the *MovementModel* class and overriding at least the *getInitialLocation* and *getPath* methods. The first method is used for getting the location where the node should be in the beginning of the simulation and the second for requesting a new path the node will use. The paths are sequences of waypoints and the nodes move from waypoint to waypoint with speed determined by the movement model. The MovementModel class also provides ability to configure and generate uniform speed and wait-time[1] distributions.

Event generators should be created to the *input* package and they must all implement the *EventQueue* interface. Using the interface the simulator engine can query when the next event is due and when the time is there, it can also ask for the event. When the event is processed, it gets a reference to all the simulated nodes so it can, e.g., create messages, setup connections or abort transfers between them by calling the right methods for the *DTNHost* objects.

Finally, the report modules, found from the *report* package, provide means for generating customizable statistics and reports of the simulation. All custom modules must extend the *Report* class and implement one or more of the event listening interfaces. A report module implementing the *MessageListener* interface will be informed of message-related events such as creating a new message or transferring it from a node to another. Modules implementing the *ConnectionListener* interface get notified when a connection between two nodes goes up or down. The *MovementListener* interface is for tracking new destinations of the moving nodes, and if none of these interfaces is enough, a report module can implement the *UpdateListener* interface and get notified every time the nodes are updated (i.e., on every time step).

[1]The time to wait still before selecting a new path

3.6 Limitations

Even though the ONE is a quite capable simulator, it naturally also has its limitations. To make simulations feasible, many real world aspects have been abstracted or even completely discarded. Some of these features are due to the innate nature of the simulator but others are possibly some day solved by future versions of the simulator.

During the simulation, the simulation time is increased in finite steps, and all nodes are moved and their message transfers are progressed for the same time. If something would have happened in the middle of the time step, e.g., a transfer had finished or a connection would have broken because of node mobility, from simulator's point of view, the event happens only before or after a full time step. The time step can be configured to be practically as small as needed, but this also makes the simulation slower. As a compromise, additional update requests can be scheduled by any module between time steps, but the simulation is still done using discrete events which can only approximate real-world behavior.

Even if the time step size is kept reasonable, simulating interactions of large amount of nodes requires quite a lot of processing power. A scenario with one thousand nodes and 0.1 second time step can still be simulated with speed of over 10 simulated seconds per second on a commodity hardware, but with larger population, and using some of the more complex routing modules, the speed can quickly drop to less than one simulated second per second. Also, the Random Access Memory (RAM) consumption, especially with the MaxProp routing module and thousands of nodes, can be multiple gigabytes. This sets limits on the size of the scenarios that can be simulated using the ONE.

Another limitation of the simulation environment is the lack of lower layer, such as physical and MAC, support. When two nodes are in the range of each other they can communicate at whatever speed is configured for connections. In real world obstacles, distance and interference affect the achievable transmission speed, but, at the moment, this is not taken into consideration. Also, the radio devices that the nodes use in simulation are now constantly turned on whereas in real world, e.g. to save battery, they are

often switched to idle mode and other devices are probed only periodically. Because of this, the contact times the ONE generates can be too optimistic in many scenarios.

The mobility models, even though fairly complex and realistic compared to simple models such as RWP, still lack many real-world details. For example, node movement using SPMBM is similar throughout the simulation time, whereas normally people behave differently depending on the time of the day. The simulation of public transportation, such as buses and trams, is also limited since they can only act as a single node and other nodes can not travel using them. However, the movement models have become more realistic with recent additions [8]. The new movement model, dubbed *Working Day Movement Model*, takes into account many factors of real human daily behavior and also evolves the way how different transport mechanisms can be used.

3.7 Results

This report did not include any simulation results from the ONE, but it has already been actively used for DTN and mobility research. For example, the first published results using the ONE [15] looked into how adding bits of realism change the connectivity patterns of wireless nodes and how different DTN routing algorithms perform in these settings. We ran over 1000 different scenarios and found out, e.g., that more realistic movement (e.g., SPMBM vs. RWP or MBM) increases substantially the delivery probability and decreases latency for most of the routing modules if the other variables are kept constant. Another study [22] focused on fragmentation in DTNs and for that both proactive and reactive fragmentation schemes were implemented to the ONE. In that study we showed that reactive fragmentation can have in many cases a positive impact on message delivery probabilities without hurting the delivery latency.

Chapter 4

Conclusions

In this report we have introduced the Opportunistic Network Environment simulator, the ONE, and described it's features, usage, and how it can be extended. The main parts of the ONE: mobility modeling, routing simulation, event generators and report modules where described in more detail and also different way of running simulations were introduced in chapter 3. The section 3.5 focused on illustrating the software architecture and how it can be used for extending the features of the simulation environment. Also, limitations of the ONE were discussed in the section 3.6. Finally, some DTN and mobility research done using the ONE was presented in the section 3.7.

The ONE, as how we see it, is now just at the beginning of it's evolution to a great tool for DTN, and also other opportunistic network paradigm, simulations. Our future work includes addressing the limitations discussed in section 3.6, exploring security issues in DTNs, and also implementing new routing algorithms, movement models, event generators and report modules. Hopefully many of the new modules will be usable as plugins for ONE so that also other research teams can easily start using them. However, many new modification are likely to need more work in the simulation internals and thus new versions of the simulation environment are likely to be released.

We also hope that ONE is taken widely into use in the DTN community and for fostering this purpose we have created a mailing list forum where users can ask questions, get answers, and share their knowledge of the ONE and DTN simulations in general. So far the ONE home page has had over

one thousand visitors from 43 different countries [9], and also the numerous questions received by e-mail about the ONE imply that multiple research institutes around the world have already taken the ONE into use.

Bibliography

[1] AT&T Research. Graphviz - Graph Visualization Software. http://www.graphviz.org/, 2008. [Online; accessed 19-May-2008].

[2] P. Ball. *Introduction to Discrete Event Simulation*. University of Strathclyde, 1996.

[3] J. Burgess, B. Gallagher, D. Jensen, and B. N. Levine. MaxProp: Routing for Vehicle-Based Disruption-Tolerant Networks. In *Proceedings of IEEE Infocom*, April 2006.

[4] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, and H .Weiss. Delay-Tolerant Networking Architecture. RFC 4838, April 2007.

[5] Dartmouth College. Community Resource for Archiving Wireless Data At Dartmouth. http://crawdad.cs.dartmouth.edu/. [Online; accessed 19-May-2008].

[6] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.

[7] Nathan Eagle and Alex (Sandy) Pentland. Reality mining: sensing complex social systems. *Personal Ubiquitous Computing*, 10(4):255–268, 2006.

[8] Frans Ekman. Mobility models for mobile ad hoc network simulations. Master's thesis, Helsinki University of Technology, May 2008.

[9] eXTReMe digital. Statistic for the ONE homepage. http://extremetracking.com/open;geo?login=akeranen. [Online; accessed 28-May-2008].

[10] Kevin Fall. A Delay-Tolerant Network Architecture for Challenged Internets. In *Proceedings of ACM SIGCOMM 2003*, pages 27–36, August 2003.

[11] Free Software Foundation. GNU General Public License. http://www.gnu.org/licenses/gpl-3.0.html. [Online; accessed 22-May-2008].

[12] Gnuplot - An Interactive Plotting Program. http://gnuplot.info/, 2008. [Online; accessed 19-May-2008].

[13] Sushant Jain, Kevin Fall, and Rabin Patra. Routing in a delay tolerant network. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 145–158, New York, NY, USA, 2004. ACM.

[14] David B Johnson and David A Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[15] Ari Keränen and Jörg Ott. Increasing Reality for DTN Protocol Simulations. Technical report, Helsinki University of Technology, Networking Laboratory, July 2007.

[16] Stuart Kurkowski, Tracy Camp, Neil Mushell, and Michael Colagrosso. A visualization and analysis tool for ns-2 wireless simulations: inspect. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 503–506, Washington, DC, USA, 2005. IEEE Computer Society.

[17] Anders Lindgren and Avri Doria. Probabilistic Routing Protocol for Intermittently Connected Networks. Internet Draft draft- lindgren-dtnrg-prophet-02, Work in Progress, March 2006.

[18] Gustavo Marfia, Giovanni Pau, Enzo De Sena, Eugenio Giordano, and Mario Gerla. Evaluating Vehicle Network Strategies for Downtown Portland: Opportunistic Infrastructure and the Importance of Realistic Mobility Models. In *Proceedings of ACM MobiOpp*, June 2007.

[19] The Network Simulator NS-2. `http://www.isi.edu/nsnam/ns/`. [Online; accessed 24-May-2008].

[20] University of Waterloo. DTNSim2 Delay-tolerant Network Simulator. `http://watwire.uwaterloo.ca/DTN/sim/`. [Online; accessed 24-May-2008].

[21] OpenJUMP - The free, Java based and open source Geographic Information System for the World. `http://openjump.org`, 2008. [Online; accessed 19-May-2008].

[22] Mikko Pitkänen, Ari Keränen, and Jörg Ott. Message Fragmentation in Opportunistic DTNs. In *Proceedings of the Second WoWMoM Workshop on Autonomic and Opportunistic Communications (AOC) 2008*. IEEE, 2008.

[23] The Rice Monarch Project. Rice Monarch Project Extensions to ns-2. `http://www.monarch.cs.rice.edu/cmu-ns.html`. [Online; accessed 19-May-2008].

[24] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S. Raghavendra. Single-copy routing in intermittently connected mobile networks. In *Proc. Sensor and Ad Hoc Communications and Networks SECON*, pages 235–244, October 2004.

[25] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S. Raghavendra. Spray and Wait: An Efficient Routing Scheme for Intermittently Connected Mobile Networks. In *Proceedings of ACM SIGCOMM Workshop on Delay-Tolerant Networking (WDTN)*, 2005.

[26] A. Vahdat and D. Becker. Epidemic routing for partially connected ad hoc networks. Technical Report CS-200006, Duke University, April 2000.

[27] Andras Varga. The OMNET++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference*, pages 319–324, Prague, Czech Republic, June 2001. SCS – European Publishing House.

Appendix A

The ONE ReadMe

The ONE is a Opportunistic Network Environment simulator which provides a powerful tool for generating mobility traces, running DTN messaging simulations with different routing protocols, and visualizing both simulations interactively in real-time and results after their completion.

Quick start
===========

Running

ONE can be started using the included one.bat (for Windows) or one.sh (for Linux/Unix) script. Following examples assume you're using the Linux/Unix script (just replace .sh by .bat for Windows).

Synopsis:
one.sh [-b] [conf-file] [run-index (count)]

Options:
  -b Run simulation in batch mode. Doesn't start GUI but prints information about the progress to terminal.

Parameters:
  conf-file: The configuration file where simulation parameters are read from.

run-index: If running in GUI mode (without -b option), you can give which run index to use in the given run. In batch mode, the last parameter is the run index count, i.e., how many runs with different run indeces are done.

Configuring
===========

All simulation parameters are given using configuration files. These files are normal text files that contain key-value pairs. Syntax for most of the variables is:
Namespace.key = value

I.e., the key is (usually) prefixed by a namespace, followed by a dot, and then key name. Key and value are separated by equals-sign. Namespaces start with capital letter and both namespace and keys are written in CamelCase (and are case sensitive). Namespace defines (loosely) the part of the simulation environment where the setting has effect on. Many, but not all, namespaces are equal to the class name where they are read. Especially movement models, report modules and routing modules follow this convention.

Numeric values use '.' as the decimal separator and can be suffixed with kilo (k) mega (M) or giga (G) suffix. Boolean settings accept "true", "false", "0", and "1" as values.

Many settings define paths to external data files. The paths can be relative or absolute but the directory separator must be '/' in both Unix and Windows environment.

Some variables contain comma-separated values, and for them the syntax is:
Namespace.key = value1, value2, value3, etc.

For run-indexed values the syntax is:
Namespace.key = [run1value; run2value; run3value; etc]
I.e., all values are given in brackets and values for different run are separated by semicolon. Each value can also be a comma-separated value.

For more information about run indexing, go to section "Run indexing".

Setting files can contain comments too. A comment line must start with "#" character. Rest of the line is skipped when the settings are read. This can be also useful for disabling settings easily.

Some values (scenario and report names at the moment) support "value filling". With this feature, you can construct e.g., scenario name dynamically from the setting values. This is especially useful when using run indexing. Just put setting key names in the value part prefixed and suffixed by two percent (%) signs. These placeholders are replaces by the current setting value from the configuration file. See the included snw_comparison_settings.txt for an example.

File "default_settings.txt" is always read and the (optional) configuration file given as parameter can define more settings or override some (or even all) settings in the default settings file. The idea is that you can define in the default file all the settings that are common for all the simulations and run different, specific, simulations using different configuration files. If your simulations don't have any common parameters (which is highly unlikely) just provide an empty default settings file. If you want to use more than one default configuration set, just create separate folders for all configuration sets and provide one default settings file for each folder.

Run indexing

Run indexing is a feature that allows you to run large amounts of different configurations using only single configuration file. The idea is that you provide an array of settings (using the syntax described above) for the variables that should be changed between runs. For example, if you want to run the simulation using five different random number generator seeds for movement models, you can define in the settings file the following:

```
MovementModel.rngSeed = [1; 2; 3; 4; 5]
```

Now, if you run the simulation using command:

```
one.sh -b my_config.txt 5
```

you would run first using seed 1 (run index 0), then another run using seed 2 etc. Note that you have to run it using batch mode (-b option) if you want to use different values. Without the batch mode flag the last parameter is the run index to use when running in GUI mode.

Run indexes wrap around: used value is the value at index (runIndex % arrayLength). Because of wrapping, you can easily run large amount of permutations easily. For example, if you define two key-value pairs:

```
key1 = [1; 2]
key2 = [a; b; c]
```

and run simulation using run-index count 6, you would get all permutations of the two values (1,a; 2,b; 1,c; 2,a; 1,b; 2,c). This naturally works with any amount of arrays. Just make sure that the smallest common nominator of all array sizes is 1 (e.g., use arrays whose sizes are primes) -- unless you don't want all permutations but some values should be paired.

Movement models

Movement models govern the way nodes move in the simulation. They provide coordinates, speeds and pause times for the nodes. The basic installation contains 5 movement models: random waypoint, map based movement, shortest path map based movement, map route movement and external movement. All models, except external movement, have configurable speed and pause time distributions. A minimum and maximum values can be given and the movement model draws uniformly distributed random values that are within the given range. Same applies for pause times. In external movement model the speeds and pause times are interpreted from the given data.

When a node uses the random waypoint movement model (RandomWaypoint), it is given a random coordinate in the simulation area. Node moves directly to the given destination at constant speed, pauses for a while, and then gets a new destination. This continues throughout the simulations and nodes move along these zig-zag paths.

Map-based movement models constrain the node movement to predefined paths. Different types of paths can be defined and one can define valid paths for all node groups. This way e.g., cars can be prevented from driving indoors or on pedestrian paths.

The basic map-based movement model (MapBasedMovement) initially distributes the nodes between any two adjacent (i.e., connected by a path) map nodes and then nodes start moving from adjacent map node to another. When node reaches the next map node, it randomly selects the next adjacent map node but chooses the map node where it came from only if that is the only option (i.e., avoids going back to where it came from). Once node has moved trough 10-100 map nodes, it pauses for a while and then starts moving again.

The more sophisticated version of the map-based movement model (ShortestPathMapBasedMovement) uses Dijkstra's shortest path algorithm to find its way trough the map area. Once a node reaches its destination, and has waited for the pause time, a new random map node is chosen and node moves there using the shortest path that can be taken using only valid map nodes.

For the shortest path based movement models, map data can also contain Points Of Interest (POIs). Instead of selecting any random map node for the next destination, the movement model can be configured to give a POI belonging to a certain POI group with a configurable probability. There can be unlimited amount of POI groups and all groups can contain any amount of POIs. All node groups can have different probabilities for all POI groups. POIs can be used to model e.g., shops, restaurants and tourist attractions.

Route based movement model (MapRouteMovement) can be used to model nodes that follow certain routes, e.g. bus or tram lines. Only the stops on the route have to be defined and then the nodes using that route move from stop to stop using shortest paths and stop on the stops for the configured time.

All movement models can also decide when the node is active (moves and can be connected to) and when not. For all models, except for the external movement, multiple simulation time intervals can be given and the nodes in that group will be active only during those times.

All map-based models get their input data using files formatted with a subset of the Well Known Text (WKT) format. LINESTRING and MULTILINESTRING directives of WKT files are supported by the parser for map path data. For point data (e.g. for POIs), also the POINT directive is supported. Adjacent nodes in a (MULTI)LINESTRING are considered to form a path and if some lines contain some vertex(es) with exactly the same coordinates, the paths are joined from those places (this is how you create intersections). WKT files can be edited and generated from real world map data using any suitable Geographic Information System (GIS) program. The map data included in the simulator distribution was converted and edited using the free, Java based OpenJUMP GIS program.

Different map types are defined by storing the paths belonging to different types to different files. Points Of Interest are simply defined with WKT POINT directive and POI groups are defined by storing all POIs belonging to a certain group in the same file. All POIs must also be part of the map data so they are accessible using the paths. Stops for the routes are defined with LINESTRING and the stops are traversed in the same order they appear in the LINESTRING. One WKT file can contain multiple routes and they are given to nodes in the same order as they appear in the file.

The experimental movement model that uses external movement data (ExternalMovement) reads timestamped node locations from a file and moves the nodes in the simulation accordingly. See javadocs of ExternalMovementReader class from input package for details of the format. A suitable, experimental converter script (transimsParser.pl) for TRANSIMS data is included in the toolkit folder.

The movement model to use is defined per node group with the "movementModel" setting. Value of the setting must be a valid movement model class name from the movement package. Settings that are common for all movement models are read in the MovementModel class and movement model specific settings are read in the respective classes. See the javadoc documentation and example

configuration files for details.

Routing modules and message creation

Routing modules define how the messages are handled in the simulation. Six different active routing modules (First Contact, Epidemic, Spray and Wait, Direct delivery, PRoPHET and MaxProp) and also a passive router for external routing simulation are included in the package. The active routing modules are implementations of the well known routing algorithms for DTN routing. See the classes in routing package for details.

Passive router is made especially for interacting with other (DTN) routing simulators or running simulations that don't need any routing functionality. The router doesn't do anything unless commanded by external events. These external events are provided to the simulator by a class that implements the EventQueue interface.

The current release includes two classes that can be used as a source of message events: ExternalEventsQueue and MessageEventGenerator. The former can read events from a file that can be created by hand, with a suitable script (e.g., createCreates.pl script in the toolkit folder), or by converting e.g., dtnsim2's output to suitable form. See StandardEventsReader class from input package for details of the format. MessageEventGenerator is a simple message generator class that creates uniformly distributed message creation patterns with configurable message creation interval, message size and source/destination host ranges.

The toolkit folder contains an experimental parser script (dtnsim2parser.pl) for dtnsim2's output (there used to be a more capable Java-based parser but it was discarded in favor of this more easily extendable script). The script requires a few patches to dtnsim2's code and those can be found from the toolkit/dtnsim2patches folder.

The routing module to use is defined per node group with the setting "router". All routers can't interact properly (e.g., PRoPHET router can only work with other PRoPHET routers) so usually it makes sense to use the same (or compatible) router for all groups.

Reports

Reports can be used to create summary data of simulation runs, detailed data of connections and messages, files suitable for post-processing using e.g., Graphviz (to create graphs) and also to interface with other programs. See javadocs of report-package classes for details.

There can be any number of reports for any simulation run and the number of reports to load is defined with "Report.nrofReports" setting. Report class names are defined with "Report.reportN" setting, where N is an integer value starting from 1. The values of the settings must be valid report class names from the report package. The output directory of all reports (which can be overridden per report class with the "output" setting) must be defined with Report.reportDir -setting. If no "output" setting is given for a report class, the resulting report file name is "ReportClassName_ScenarioName.txt".

All reports have many configurable settings which can be defined using ReportClassName.settingKey -syntax. See javadocs of Report class and specific report classes for details (look for "setting id" definitions).

Host groups

A host group is group of hosts (nodes) that shares movement and routing module settings. Different groups can have different values for the settings and this way they can represent different types of nodes. Base settings can be defined in the "Group" namespace and different node groups can override these settings or define new settings in their specific namespaces (Group1, Group2, etc.).

The settings

There are plenty of settings to configure; more than is meaningful to present here. See javadocs of especially report, routing and movement model classes for details. See also included settings files for examples. Perhaps the most important settings are the following.

Scenario settings:

Scenario.name
Name of the scenario. All report files are by default prefixed with this.

Scenario.simulateConnections
Should connections be simulated. If you're only interested in movement modeling, you can disable this to get faster simulation. Usually you want this to be on.

Scenario.updateInterval
How many seconds are stepped on every update. Increase this to get faster simulation, but then you'll lose some precision. Values from 0.1 to 2 are good for simulations.

Scenario.endTime
How many simulated seconds to simulate.

Scenario.nrofHostGroups
How many hosts group are present in the simulation.

Host group settings (used in Group or GroupN namespace):

groupID
Group's identifier (a string or a character). Used as the prefix of host names that are shown in the GUI and reports. Host's full name is groupID+networkAddress.

nrofHosts
Number of hosts in this group.

transmitRange
Range (meters) of the hosts' radio devices.

transmitSpeed
Transmit speed of the hosts' radio devices (bytes per second).

movementModel
The movement model all hosts in the group use. Must be a valid class (one that is a subclass of MovementModel class) name from the movement package.

waitTime
Minimum and maximum (two comma-separated decimal values) of the wait time interval (seconds). Defines how long nodes should stay in the same place after reaching the destination of the current path. A new random value within the interval is used on every stop. Default value is 0,0.

speed
Minimum and maximum (two comma-separated decimal values) of the speed interval (m/s). Defines how fast nodes move. A new random value is used on every new path. Default value is 1,1.

bufferSize
Size of the nodes' message buffer (bytes). When the buffer is full, node can't accept any more messages unless it drops some old messages from the buffer.

router
Router module which is used to route messages. Must be a valid class (subclass of Report class) name from routing package.

activeTimes
Time intervals (comma-separated simulated time value tuples: start1, end1, start2, end2, ...) when the nodes in the group should be active. If no intervals are defined, nodes are active all the time.

msgTtl
Time To Live (simulated minutes) of the messages created by this host group. Nodes (with active routing module) check every one minute whether some of their messages' TTLs have expired and drop such messages. If no TTL is defined, infinite TTL is used.

Group and movement model specific settings (only meaningful for certain movement models):

pois
Points Of Interest indexes and probabilities (comma-separated index-probability tuples: poiIndex1, poiProb1, poiIndex2, poiProb2, ...). Indexes are integers and probabilities are decimal values in the range of 0.0-1.0. Setting defines the POI groups where the nodes in this host group can choose destinations from and the probabilities for choosing a certain POI group. For example, a (random) POI from the group defined in the POI file1 (defined with PointsOfInterest.poiFile1 setting) is chosen with the probability poiProb1. If the sum of all probabilities is less than 1.0, a probability of choosing any random map node for the next destination is (1.0 - theSumOfProbabilities). Setting can be used only with ShortestPathMapBasedMovement -based movement models.

okMaps
Which map node types (refers to map file indexes) are OK for the group (comma-separated list of integers). Nodes will not travel trough map nodes that are not OK for them. As default, all map nodes are OK. Setting can be used with any MapBasedMovent -based movement model.

routeFile
If MapRouteMovement movement model is used, this setting defines the route file (path) where the route is read from. Route file should contain LINESTRING WKT directives. Each vertex in a LINESTRING represents one stop on the route.

routeType
If MapRouteMovement movement model is used, this setting defines the routes type. Type can be either circular (value 1) or ping-pong (value 2). See movement.map.MapRoute class for details.

Movement model settings:

MovementModel.rngSeed
The seed for all movement models' random number generator. If the seed and all the movement model related settings are kept the same, all nodes should move the same way in different simulations (same destinations and speed & wait time values are used).

MovementModel.worldSize
Size of the simulation world in meters (two comma separated values:
width, height).

PointsOfInterest.poiFileN
For ShortestPathMapBasedMovement -based movement models, this setting defines the WKT files where the POI coordinates are read from. POI coordinates are defined using the POINT WKT directive. The "N" in the end of the setting must be a positive integer (i.e., poiFile1, poiFile2, ...).

MapBasedMovement.nrofMapFiles
How many map file settings to look for in the settings file.

MapBasedMovement.mapFileN
Path to the Nth map file ("N" must be a positive integer). There must be at least nrofMapFiles separate files defined in the configuration files(s). All map files must be WKT files with LINESTRING and/or MULTILINESTRING WKT directives. Map files can contain POINT directives too, but those are skipped. This way the same file(s) can be used for both POI and map data. By default the map coordinates are translated so that the upper left corner of the map is at coordinate point (0,0). Y-coordinates are mirrored before translation so that the map's north points up in the playfield view. Also all POI and route files are translated to match to the map data transformation.

Report settings:

Report.nrofReports
How many report modules to load. Module names are defined with settings "Report.report1", "Report.report2", etc. Following report settings can be defined for all reports (using Report name space) or just for certain reports (using ReportN name spaces).

Report.reportDir
Where to store the report output files. Can be absolute path or relative to the path where the simulation was started. If the directory doesn't exists, it is created.

Report.warmup
Length of the warm up period (simulated seconds from the start). During the warm up the report modules should discard the new events. The behavior is report module specific so check the (java)documentation of different report modules for details.

Event generator settings:

Events.nrof
How many event generators are loaded for the simulation. Event generator specific settings (see below) are defined in EventsN namespaces (so Events1.settingName configures a setting for the 1st event generator etc.).

EventsN.class
Name of the generator class to load (e.g., ExternalEventsQueue or MessageEventGenerator). The class must be found from the input package.

For the ExternalEventsQueue you must at least define the path to the external events file (using setting "filePath"). See input.StandardEventsReader class' javadocs for information about different external events.

Other settings:

Optimization.randomizeUpdateOrder
Should the order in which the nodes' update method is called be randomized. Call to update causes the nodes to check their connections and also update their routing module. If set to false, node update order is the same as their network address order. With randomizing, the order is different on every time step.

GUI
===

The GUI's main window is divided into three parts. The main part contains the playfield view (where node movement is displayed) and simulation and GUI control and information. The right part is used to select nodes and the lower part is for logging and breakpoints.

The main part's topmost section is for simulation and GUI controls. The first field shows the current simulation time. Next field shows the simulation speed (simulated seconds per second). The following four buttons are used to pause, step, fast forward, and fast forward simulation to given time. Pressing step-button multiple times runs simulation step-by-step. Fast forward (FFW) can be used to skip uninteresting parts of simulation. In FFW, the GUI update speed is set to a large value. Next drop-down is used to control GUI update speed. Speed 1 means that GUI is updated on every simulated second. Speed 10 means that GUI is updated only on every 10th second etc. Negative values slow down the simulation. The following drop-down controls the zoom factor. The last button saves the current view as a png-image.

Middle section, i.e., the playfield view, shows the node placement, map paths, node identifiers, connections among nodes etc. All nodes are displayed as small rectangles and their radio range is shown as a green circle around the node. Node's group identifier and network address (a number) are shown next to each node. If a node is carrying messages, each message is represented by a green or blue filled rectangle. If node carries more than 10 messages, another column of rectangles is drawn for each 10 messages but every other rectangle is now red. You can center the view to any place by clicking with mouse button on the play field. Zoom factor can also be changed using mouse wheel on top of the playfield view.

The right part of main window is for choosing a node for closer inspection. Simply clicking a button shows the node info in main parts lower section. From there more information can be displayed by selecting one of the messages the node is carrying (if any) from the drop-down menu. Pressing the "routing info" button opens a new window where information about the routing module is displayed. When a node is chosen, the playfield view is

also centered on that node and the current path the node is traveling is shown in red.

Logging (the lowest part) if divided to two sections, control and log. From the control part you can select what kind of messages are shown in the log. You can also define if simulation should be paused on certain type of event (using the check boxes in the "pause" column). Log part displays time stamped events. All nodes and message names in the log messages are buttons and you can get more information about them by clicking the buttons.

Toolkit
=======

The simulation package includes a folder called "toolkit" that contains scripts for generating input and processing the output of the simulator. All (currently included) scripts are written with Perl (http://www.perl.com/) so you need to have it installed before running the scripts. Some post processing scripts use gnuplot (http://www.gnuplot.info/) for creating graphics. Both of the programs are freely available for most of the Unix/Linux and Windows environments. For Windows environment, you may need to change the path to the executables for some of the scripts.

getStats.pl
This script can be used to create bar-plots of various statistics gathered by the MessageStatsReport -report module. The only mandatory option is "-stat" which is used to define the name of the statistics value that should be parsed from the report files (e.g., "delivery_prob" for message delivery probabilities). Rest of the parameters should be MessageStatsReport output filenames (or paths). Script creates three output files: one with values from all the files, one with the gnuplot commands used to create the graphics and finally an image file containing the graphics. One bar is created to the plot for each input file. The title for each bar is parsed from the report filename using the regular expression defined with "-label" option. Run getStats.pl with "-help" option for more help.

ccdfPlotter.pl
Script for creating Complementary(/Inverse) Cumulative Distribution Function plots (using gluplot) from reports that contain time-hitcount-tuples. Output filename must be defined with the "-out" option and rest of the parameters should be (suitable) report filenames. "-label" option can be used for defining label extracting regular expression (similar to one for the getStats script) for the legend.

createCreates.pl
Message creation pattern for the simulation can be defined with external events file. Such a file can be simply created with any text editor but this script makes it easier to create a large amount of messages. Mandatory options are the number of messages ("-nrof"), time range ("-time"), host address range ("-hosts") and message size range ("-sizes"). The number of messages is simply an integer but the ranges are given with two integers with a colon (:) between them. If hosts should reply to the messages that they receive, size range of the reply messages can be defined with "-rsizes" option. If a certain random number generator seed should be used, that can be defined with "-seed" option. All random values are drawn from a uniform distribution with inclusive minimum value and exclusive maximum value. Script outputs commands that are suitable for external events file's contents. You probably want to redirect the output to some file.

dtnsim2parser.pl and transimsParser.pl
These two (quite experimental) parsers convert data from other programs to a form that is suitable for ONE. Both take two parameters: input and output file. If these parameters are omitted, stdin and stdout are used for input and output. With "-h" option a short help is printed.
dtnsim2parser converts dtnsim2's (http://watwire.uwaterloo.ca/DTN/sim/) output (with verbose mode 8) to an external events file that can be fed to ONE. The main idea of this parser is that you can first create a connectivity pattern file using ONE and ConnectivityDtnsim2Report, feed that to dtnsim2 and then observe the results visually in ONE (using the output converted with dtnsim2parser as the external events file).
transimsParser can convert TRANSIM's (http://transims-opensource.net/) vehicle snapshot files to external movement files that can be used as an input for node movement. See ExternalMovement and ExternalMovementReader classes for more information.