TRAILS, a Toolkit for Efficient, Realistic and Evolving Models of Mobility, Faults and Obstacles in Wireless Networks *

Ioannis Chatzigiannakis, Athanasios Kinalis, Georgios Mylonas,
Sotiris Nikoletseas, Grigorios Prasinos and Christos Zaroliagis
Computer Technology Institute (CTI),
N. Kazantzaki Str., 26500 Patras, Greece.
Tel: +30 2610 960324, Fax: +30 2610 960442.
Department of Computer Engineering and Informatics,
University of Patras, 26500 Patras, Greece. Tel: +30 2610 960459
{ichatz,kinalis,mylonasg,nikole,green,zaro}@cti.gr

Abstract

*We present a new simulation toolkit called TRAILS (Toolkit for Realism and Adaptivity In Large-scale Simulations), which extends the **ns-2** simulator by adding important functionality and optimizing certain critical simulator operations. The added features provide the tools to study wireless networks of high dynamics. TRAILS, facilitates the implementation of advanced mobility patterns, obstacle presence and disaster scenarios, and failures injection that can dynamically change throughout the execution of the simulation. Moreover, we define a set of utilities that enhance the use of **ns-2** This functionality is implemented in a simple and flexible architecture, that follows design patterns, object oriented and generic programming principles, maintaining a proper balance between reusability, extendability and ease of use. We evaluate the performance of TRAILS and show that it offers significant speed-ups (at least 4 times faster) regarding the execution time of **ns-2** in certain important, common wireless settings. Our results also show that this is achieved with minimum overhead in terms of memory usage.*

*This work has been partially supported by the IST Programme of the European Union under contract number IST-2005-15964 (AEOLUS), the ICT Programme of the European Union under contract number IST-2008-215270 (FRONTS), the PYTHAGORAS Programme under the European Social Fund (ESF) and Operational Program for Educational and Vocational Training II (EPEAEK II).Also, by the Programme **PENED** under contract number 03ED568, co-funded 75% by European Union – European Social Fund (ESF), 25% by Greek Government – Ministry of Development – General Secretariat of Research and Technology (GSRT), and by Private Sector, under Measure 8.3 of O.P. Competitiveness – 3rd Community Support Framework (CSF).

1. Introduction

Research in mobile ad hoc networks (MANET), wireless sensor networks (WSN) and vehicular ad hoc networks (VANET) has grown both in volume and significance over the last few years, amplified by the fact that several pilot applications have surfaced, especially in the case of wireless sensor networks. Currently, a number of simulators have been developed and extended to allow the modeling and simulation of MANETs, WSNs and VANETs. Most notable examples of such simulators are **ns-2**, **OMNeT++** and **OPNET Modeler**.

Even though such simulators provide the basic framework required to simulate a generic wireless network, lack the capacity to produce realistic simulations that capture real world scenarios. In many applications wireless nodes move in complex ways, may have to avoid obstacles and eventually may cease to function due to enviromental functions. Also, researchers make *a lot* of simplifying assumptions in order to reduce the simulation complexity. Such oversimplifications are sometimes required in order to conduct experiments with a great number of nodes in reasonable time, but most frequently they are imposed by the lack of functionality in the simulation tools. In this way the credibility of such research works is weakened. A discussion of common pitfalls and shortcomings in the simulation of wireless networks in general is included in [15].

One of the most widely used simulators for research in MANETs, WSNs and VANETs, is **ns-2**. Despite its popularity, **ns-2** lacks several "advanced" features that allow more detailed and realistic simulation of the aforementioned categories of networks. Users of the simulator usually re–implement the above functionality as needed, thus increasing the effort required for implementing and eval-

uating a protocol. Also, the direct comparison of results between scientific papers is difficult, since implementation differences may affect the results, as noted in [15]. We believe that the following list of functional requirements are necessary to perform simulations that better capture the dynamic characteristics of previously mentioned systems.
Realistic Models: Researchers often assume uniform distribution of nodes, simple random walks, ignore failures, etc. Simulators should facilitate the implementation of more realistic models. **Reactive Mobility:** Studying only predetermined motion or randomized walks is not realistic enough. In real world scenarios, nodes will change their movement based on simulator events thus online computation of complex mobility patterns is required. **Dynamic Obstacles:** Realistic simulations should allow the simulation of dynamic phenomena, such as natural disasters (e.g. floods and forest fires). These phenomena obstruct movement and communications, and evolve through time changing the topology. **Advanced Statistics:** Gathering statistics is a twofold process; not only do we need statistics of network and protocol metrics at the end of simulation, but it is also desirable to have simple statistics, such as number of neighbors and remaining energy, available at the nodes during the simulation. **Usability & Reusability:** Simulator constructs need to be easily configurable, extensible and reusable. Currently in **ns-2** certain functions are spread out in several components, developers have to insert arbitrary code in core components, while protocol logic is often partly implemented in OTCL. Furthermore, the codebase does not always respect object oriented principles and, for historical reasons, the use of STL is avoided. **Efficiency in large-scale simulations:** Since **ns-2** is notorious for the amount of resources it requires, any new functionality should attempt to minimize memory and CPU usage to allow the study of complex systems.

In this work we present a new simulation framework specifically designed to address the needs we described above. Our framework extends the **ns-2** simulator adding several important functionalities and optimizing certain critical simulator operations. It provides provides three main functionalities: a) *advanced mobility*, b) *dynamic physical and communication obstacles*, and c) *advanced node failure models*. We provide a ready-to-use library that implements popular mobility models, configurable obstacles and varying failure models. At the same time we provide easy to use scripts that allow quickly setting up a simulation scenario using our advanced features. We keep the modifications to **ns-2** core minimal and preserve its current functionality.

2. Previous Work

Several other attempts have been made to improve support for the aforementioned categories of wireless networks in **ns-2**. BonnMotion [1] is a Java software which creates and analyzes mobility scenarios, serving as a tool for the investigation of mobile ad hoc network characteristics. The scenarios can be exported for the simulators **ns-2** and GlomoSim/QualNet. Also, [4] and [16] are two examples of environments that provide features for defining mobility scenarios inside urban settings, more realistic and configurable than the Manhattan model. In contrast to our approach all these approaches provide *precomputed* modeling, a scenario file is generated by the tool containing commands that define the movement. We allow the dynamic computation of a mobile node's next position based on its current position and other factors such as evolving obstacles.

Another attempt at modifying **ns-2** to better support WSNs is SensorSim [11]. SensorSim provides an abstract sensing model, by assuming that certain nodes operate as event generators that broadcast events at a configurable range and rate. We do not implement explicitly a sensing model although we provide the functionality of alerting nodes about nearby obstacles; in scenarios where obstacles are also sources of events, e.g. floods, forest fires, users can exploit this functionality to generate application specific events.

nsMiracle [2] is another extension framework for **ns-2**, that presents a new architectural perspective by allowing cross layer protocol designs. The main abstraction provided is the notion of a message channel where produced messages may be delivered in a number of consumers. nsMiracle tries to minimize interference to **ns-2** core components and relies on a modular architecture that supports many modules at each layer. Our approach is similar, we also use event producer - consumer pairs but we mostly rely on simple method calls for communication between our components.

The approach followed in most research works including obstacles in MANET simulations is to use maps of towns or university campuses and model buildings and other constructions as obstacles that block either mobility or radio communication, or both, i.e., in most cases only static obstacles cases are considered. In [13] a framework for defining such structures that affect both mobility and radio communications in **ns-2** is described.

Regarding obstacle placement in WSN, in [10] a systematic and generic obstacle model to be used in simulations of wireless sensor networks is proposed, along with a categorization of the obstacles' types, based on a number of certain criteria (such as shape, size, etc.). The advantage of using such a simple model is that it can be easily parameterized and subsequently implemented inside network simulators. Furthermore, the obstacle model provided includes a definition of *probabilistic* dynamic obstacles.

Regarding disaster scenarios modeling, there are two general directions: one for modeling conditions in the field

during the disaster event and one *after* the disaster strikes. The first approach includes the simulation of natural disasters, like wildfires and floods, and the way they evolve through time, spreading throughout the network field. Most of the relevant research focuses on the modeling of the natural disaster itself, and in general there exist very few cases that study the behavior of wireless networks under such conditions. In most cases cellular automata are used to model the spreading of the phenomenon, using mathematical formulas that try to model the interaction of parameters such as the speed and direction of wind, type of terrain, humidity and temperature, in the case of wildfires. The second approach is modeling the mobility of units that serve as a means of disaster relief and management, being the ones that form the ad hoc network. In [5] the mobility of paramedic units after a disaster event takes place is studied, pointing out the importance of the model used and the special kind of interactions between these units in such a setting.

Failures injection is a very important subject when discussing theoretically about distributed systems in general. However, in most research regarding MANET, WSN and VANET not much attention has been given to this specific subject up till now. Important aspects of failure occurrence, such as when a failure happens, the type of failures (deterministic/random/stochastic, temporary/permanent), the possibility of malicious failures due to attacks, etc., have not been satisfactorily investigated. Furthermore, in research that includes faults injection, and following the general paradigm used also in node placement and mobility, failures are injected in the network in a completely randomized fashion over the network field [9]. In contrast to the modeling of mobility, placement and obstacles, there has not been a corresponding generic approach proposed for failures to be used in network simulators. However, in WSN there has been an amount of work modeling failures in sensor components, which produce wrong sensed values and which are similar to byzantine faults injection.

3. Architecture

The high-level architecture of our toolkit consists of four core components: the *mobility domain*, the *obstacles domain*, the *topology domain*, the *failures domain* and the *statistics and auditing domain*. These four components are loosely coupled so that they can be easily removed, without affecting the functionality of the other components. In addition to these, we implement a series of helper components that enhance and expose simulator specific functionality. The components `ChannelConnector`, `MobilityNodeEx`, `TopologyConnector` and `EventConnector` provide access and extend the functionality of wireless channel, mobile node, topography and scheduler components of **ns-2**, the extensions are described in greater detail in Section 4. An overview of this architecture is given in Fig. 1.

Mobility Domain. The *Mobility Domain* encapsulates the required functionality for implementing different mobility patterns. It maintains the `MotionSpace` class that represents the area where the mobile hosts will move around and the `NodeLocation` class that holds the position of the nodes. The actual movement of nodes is governed by the `MobilityModel` class that models the movement of the nodes in a simulation. Encapsulating mobility context and mobility behavior in different classes allows each node to use a different mobility model that can be selected at runtime. Therefore, the simulator's end-user can assign different `MobilityModel` classes to each simulated node. Finally, this domain also includes the class `NetObserver` that monitors the local network and produces statistics regarding the state of the neighboring nodes. Such statistics can be used to develop adaptive network protocols. Also they can be accessed by the `MobilityModel` class in order to implement adaptive mobility patterns that are context aware (e.g., see [7]). The toolkit already defines a set of network observers (e.g., that monitor average neighborhood energy, local density etc.) and it is easy to extend it to monitor other network metrics. The architecture used allows each node to use a different set of active network observing classes that can be selected at runtime. Again, the simulator's end-user can assign different `NetObserver` classes to each simulated node. The interconnection of the components is given in Figure 2, where n is the total number of simulated nodes, k is the total number of mobility models and m is the total number of network observers assigned to the nodes.

Obstacles Domain. The *Obstacles Domain* implements the framework for simulating obstacles that affect the area where the network nodes will move and communicate. Our toolkit supports two types of obstacles: physical obstacles and communication obstacles. The physical obstacles prevent the physical presence of the nodes while communication obstacles cause disruption to wireless communication media. The `ObstacleRegistry` class holds the information related to all the obstacles that are present in the simulated area and coordinates the simulation of the different obstacle models. Since a physical obstacle may affect the movement of the mobile nodes, the `ObstacleRegistry` class communicates the geometric structures that represent an obstacle to the `MotionSpace` class.

The actual simulation of an obstacle model is implemented in the `ObstacleModel` class which holds the state representing a single obstacle. An interesting feature of this architecture is that it allows the simultaneous simulation of multiple (and possibly different) obstacle models. Also note that since obstacles may evolve

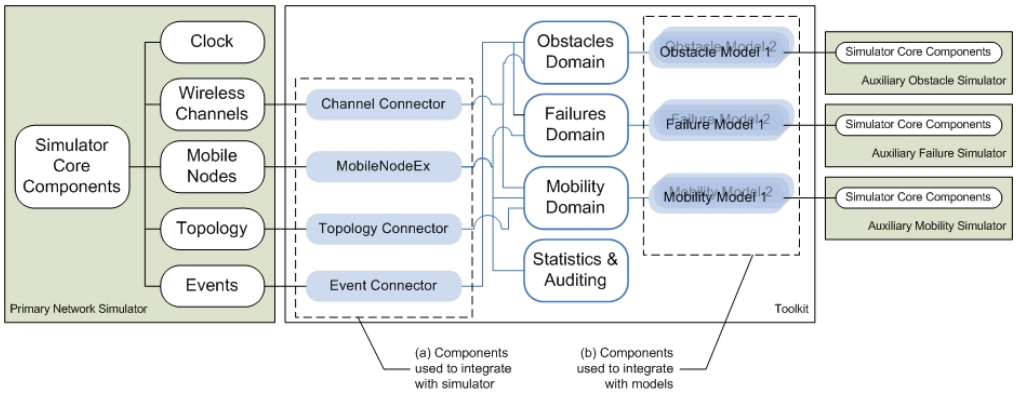


Figure 1. The architecture of our toolkit in relation to a generic simulator

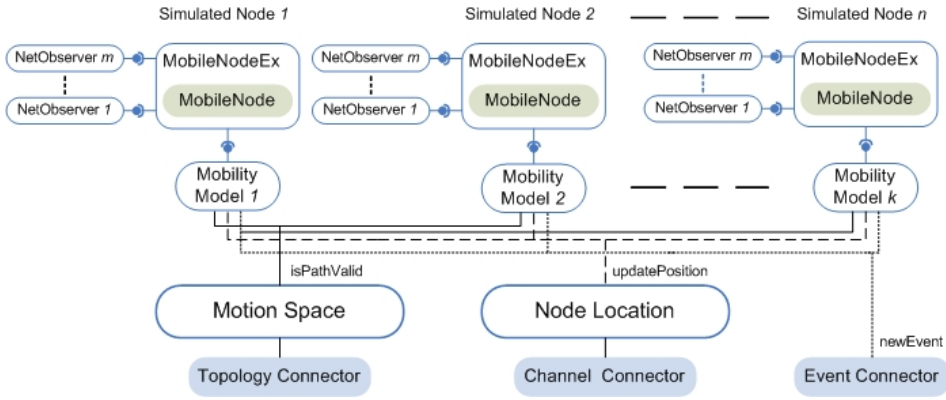


Figure 2. The components of the Mobility Domain and their interconnection

over time, an `ObstacleModel` class may interface to the `EventConnector` and recalculate its state. Essentially this means that obstacles are not necessarily present throughout the duration of a simulation experiment, but may appear in a random or deterministic fashion for a period of time and then disappear. The toolkit already defines a wide range of obstacle models (e.g., rectangular shaped, circular, crescent, ring shaped, stripe shaped etc.) and it is easy to extend it by defining other obstacle models.This allows operating auxiliary obstacle simulators that advance in parallel to the primary network simulation. The interconnection of the components is given in Figure 3, where o is the total number of simulated obstacle models.

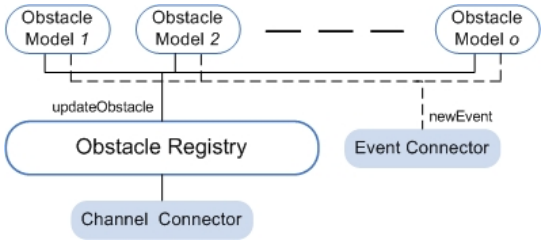


Figure 3. Obstacle Domain

Failures Domain. The *Failures Domain* provides a framework for simulating failures that affect the network nodes. Our toolkit supports two types of failures: node based and location based. The node based failures affect specific network nodes while location based failures affect all network nodes that are positioned within a particular region of the simulated area. Node failures constitute only halting errors, not Byzantine errors. The location based failures are translated to node based failures by issuing queries to the `NodeLocation` class in order to locate all the nodes affected by the area specific failure. Since failures may also result from the appearance of a physical obstacle, the `FailureRegistry` class observes the state of the `ObstacleRegistry` class. This integration of the two domains further improves the realism of the simulator.

The actual simulation of a failure model is done in a way similar to the *Obstacles Domain*, and is implemented in the `FailureModel` class (see Sec. 4). The toolkit already defines a variety of failure models (e.g., random uniform, non-uniform etc.) and it is easy to extend by defining other fail-

ure models. As in the case with obstacle models, it allows the simulation of multiple (and possibly different) failure models. Thus we can operate auxiliary failure simulators in parallel to the primary network simulation. The interconnection of the components is given in Figure 4, where f is the total number of simulated failure models.

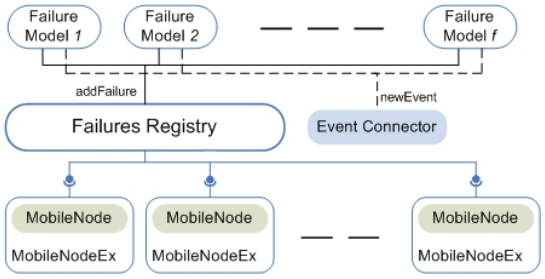


Figure 4. Failures Domain

Statistics & Auditing Domain. The *Statistics & Auditing Domain* provides two main features: a) to record scenario specific statistics, that can be accessed during simulation time by the simulated protocol, such as the number of messages sent, the number of neighbors of each node, etc. b) to encapsulate the tracing capabilities of **ns-2** allowing more transparent use of trace objects. *Statistics & Auditing Domain* provides mostly support functionality to the other main components of our framework and also to the end user. The `Monitor` class defines a calculator for a particular metric; the actual measurements are taken either by periodically polling objects or by observing the changes in the state of other objects and data can then be exposed to other parts of the simulator or logged using the `Auditor` class. The `Auditor` class is responsible for providing the mechanisms to record in persistent storage important data regarding the simulation that can be later used by the end user to extract results and conclusions.

4. Implementation & Integration with ns-2

The implementation of our toolkit is done in C/C++ and uses a variety of software design patterns [12]. We decided to follow a broad set of design patterns to speed up the development process based on proven development paradigms. This also *improves the code readability* for coders who are familiar with the particular patterns. It is our goal to provide a toolkit with high usability so that other researchers can easily implement their mobility, obstacles and failure models. Also, we believe that this *improves reusability* and allows simulator implementors to integrate our toolkit by making a small number of changes to specific parts of the existing code and within a short period of time.

Our toolkit is implemented around four components that contain the core functionality for (i) mobility, (ii) obstacles, (iii) failures and (iv) statistics and auditing. Each component is comprised of one or more singleton classes, that provide the core functionality of each component, and a number of normal helper classes that handle per node specific functionality. Also, we implement controllers in TCL that create and parameterize these components. An overview of our approach can be seen in Figure 5.

Modification of the ns-2 core and Integration. Some of the features we provide require interactions with the core components of the simulator. The most heavily affected **ns-2** object is the `MobileNode` class. In order to maintain compatibility with existing protocols that store operational parameters in `MobileNode` we extend it. The extension, called `MobileNodeEx`, exposes the exact same functionality as the original `MobileNode` class. In this extended class we add our specific hooks that are used by the components of our toolkit.

We introduce new state keeping functionality, required to implement node states such as sleep or failure. Note, that the sleep or awake state is already handled in several other places in **ns-2** (`EnergyModel`, `WirelessPhy`), we encapsulate that functionality and expose a unified interface to access those functions. Also, we keep extended mobility state such as the `MobilityModel` used, and provide functionality to enable or disable node mobility. `MobileNodeEx` makes sure that updating the position of a node is done without violating the constraints imposed by the `MotionSpace` object and if that happens, it is responsible to notify the `MobilityModel` about the violation. Furthermore, `MobileNodeEx` maintains a callback list of functions or functors that are used to gather per node statistics. These constructs enable the collection of per node statistics such as fine-grained monitoring of the energy status. `MobileNodeEx` implements statistics gathering functions measuring the number of neighboring nodes, the average energy of the neighbors and the average packet traffic rate in a node. Additionally, we modify the `command` method of `MobileNodeEx` in order to expose the extra functionality to TCL code and allow the proper setup of simulations. The rest of the components of our framework are implemented in an independent way using the functionality provided mainly by the `MobileNodeEx` class.

Furthermore, we integrate the `MotionSpace` class with the `Topography` object of **ns-2** through the `TopologyConnector` object. Currently, the `Topography` object maintains simplistic network area information, i.e., only the dimensions of a rectangle. We encapsulate this information in `MotionSpace` and add support for marking areas as unaccessible in order to model obstacles. Where required we use the geometry library CGAL [**?**], in a manner transparent to the end user.

Another important issue in **ns-2** is simulation event management. Currently, **ns-2** supports recurring events through

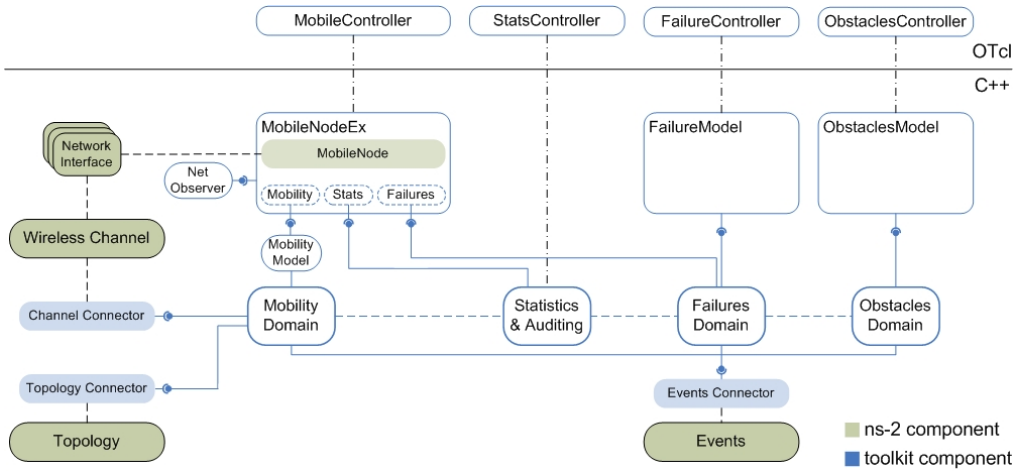


Figure 5. NS-2 components, our extensions and the coupling of components

the `TimerHandler` class. However, using this functionality requires noteworthy programming effort. We implement the `EventConnector` class that encapsulates and extends `TimerHandler` functionality, interfaces to the **ns-2** `Scheduler` class and handles the management of many events thus eliminating the need for the user to manage multiple timers. `EventConnector` is used to schedule execution of a callback function after a desired interval and supports periodic or single invocation of the callback.

Additionally, we reimplement the `WirelessChannel` class of **ns-2** using a more efficient algorithm for calculating which nodes can receive a message according to the signal strength of the transmission. More specifically, we delegate all node location management to the `NodeLocation` class which already implements an efficient mechanism based on k-d trees [6]. We substitute all relevant methods (`getAffectedNodes()`, `addNodeToList()`, `removeNodeFromList()`) with calls to `NodeLocation` methods, while leaving the rest of the functionality in place. In this manner, the `WirelessChannel` class becomes much simpler and adding new location data structures is straightforward. Also, our implementation of `getAffectedNodes` respects the status of the dead nodes and does not try to deliver messages to them. This allows to speed up the simulation in dense networks and mitigate the overhead of the online mobility models and dynamic obstacles.

Mobility Domain. The `MotionSpace` class defines the region of possible locations where nodes can move to. It interacts with the *Obstacle Domain* so that obstacles that block movement are registered and the regions occupied by these obstacles are characterized as inaccessible. In order to speed up the processing of such queries, we store the information regarding obstacles using a k-d binary tree [6], for simple shapes or CGAL geometry library for more complex shapes. Based on this implementation, in the worst case, a query on the k-d tree requires $O\left(\sqrt{e}+k\right)$, where e is the total number of obstacles and k the number of obstacles reported. Since new obstacles may be added during a simulation, or existing ones may be modified or even removed, the `MotionSpace` class observes the state of the `ObstacleRegistry` class. This is implemented by following the observer pattern that is based on the principle of implicit invocation. Whenever an obstacle is added, modified or removed, the `ObstacleRegistry` raises a new event and the `MotionSpace` class receives a callback to update the k-d binary tree. The operation of updating a balanced k-d binary tree by inserting or deleting an obstacle takes $O\left(\log e\right)$. If the shape of an obstacle needs to be modified (e.g., grow or shrink), we first delete the obstacle and then insert the new shape. In case the obstacles modeling is disabled (for a particular simulation), no events are raised and consecutively no updates to the k-d binary tree are performed.

The `MobilityModel` class represents a motion pattern for the mobile nodes and when mobility is enabled a `MobileNodeEx` object is associated with an instance of this class. The `MobilityModel` class defines an interface that `MobileNodeEx` calls when the position of the node needs to be updated. The most important function is `updatePosition` that sets the new target position and also defines a new speed for the node. This approach is quite simple yet allows intricate mobility patterns to be implemented. Implementing a new mobility pattern is straightforward, implementors can directly extend the `MobilityModel` class or a derivative of it if a similar pattern is already implemented. As usual `MobilityModel` implements a `command` method that is required for exposing state and allowing the parameterization of the class through TCL.

Obstacles Domain. The `ObstacleModel` class implements the simulation of a given obstacle. In effect `ObstacleModel` describes a geometric shape and is responsible for maintaining and updating this state. Each `ObstacleModel` is automatically registered with the `ObstacleRegistry` singleton class, which is responsible for observing the state of obstacles and update the `MotionSpace` when required.

Because these basic obstacle classes are fairly simple, they can be easily parameterized and efficiently implemented. Of course, such classes can be combined to produce more complicated obstacles and using our framework one can implement such obstacles both easily and efficiently. An example of a more complicated obstacle scheme implementation and a proof of the validity of our approach, is an obstacle class used to model natural disasters, namely wildfires. In the related research work, wildfires are modeled with the help of cellular automata. The network field is usually broken into equally-sized square cells (and in some cases in hexagon cells, like in cellular phone network models) and using transitions derived from well-known mathematical models for wildfires (the most frequently used being Rothermel's model) and a set of starting points and other parameters, the spreading of a fire is simulated. In our work, using our architecture we can define an `ObstacleModel` class that implements the logic of such a cellular automaton using a simpler `ObstacleModel` class as a basis. When the simulated phenomenon starts spreading, this `ObstacleModel` class spawns new obstacles that correspond to areas where the phenomenon has spread. Also, since wildfires burn out after a certain period of time, each such obstacle has a limited lifetime after which it is removed from the `ObstacleRegistry`. As usual, `ObstacleModel` implements a `command` method that is required for exposing state towards TCL allowing parameterization.

Failures Domain. The `FailureModel` class implements the simulation of a given failure model. `FailureModel` does not necessarily keep state of its own nor it exposes any global functionality. A simple stochastic `FailureModel` can use the information of other **ns-2** objects to select a node and then use the functionality exposed by `MobileNodeEx` to terminate the execution of the node. Periodic rescheduling is achieved with the help of the `EventConnector`. A more complex behavior is also based on the observer pattern; it registers to the `ObstacleRegistry` and injects failures to nodes that are affected by the obstacle's change. Note that in this way only halting errors of the nodes, either transient or permanent, are implemented. Currently, it is not within our plans to implement Byzantine behavior or wireless link failures.

Statistics & Auditing. Extracting results from a simulation is usually a tedious process where large trace files need to be parsed to extract and aggregate metrics. As described earlier we provide the functionality to implement custom monitors that register such metrics. `Monitor` classes can be programmed to calculate the value of a metric and to periodically report that value. We implement several monitor classes; their purpose is to gather network wide statistics that are easily obtainable without interfering with the routing protocol operation. Such monitors record the number of transmitted and dropped packets and the average node energy. Monitors are initialized and parameterized by a TCL control class.

The auditing functionality is implemented by the `Auditor` singleton class whose purpose is to encapsulate **ns-2** trace objects. The `Auditor` class acts as a container for multiple trace objects and exposes a unified interface to record information in a trace file. More specifically, `Auditor` implements a set of C++ output streams that correspond to each trace file, this abstraction is more natural to C++ programmers. Alternatively, `Auditor` also implements `printf` like functionality that is compatible with the current methodology of **ns-2**. The `Auditor` class is parameterized through a TCL controller that can be also used to create new trace objects.

Integration with TCL. Most of the components we described above implement a significant number of functions to interact with TCL. In this way a whole new array of configuration options is available to the end user that need to be properly adjusted to simulate a network realistically. This situation however, poses another barrier to inexperienced users of the toolkit thus reducing the usability of our framework. For this reason we develop a set of OTCL classes with the sole purpose to handle the initialization of a simulation scenario.

We define the `Scenario` OTCL class that encapsulates all scenario setup details and handles the initialization of the rest of the components. `Scenario` class initially creates the `Simulator` object required to interact with **ns-2** as well as create topography and trace objects and handle all the usual initialization routines. Moreover, `Scenario` handles the creation of new mobile nodes ensuring the proper initialization of our toolkit. The exact parameters governing the behavior of the simulator (e.g. transmission power, radio propagation module etc.) can be loaded through a predefined profile file. Profile files contain simple configuration variables and their value. We provide a set of ready to use profile files that capture the radio and energy characteristics of wireless sensor and mobile ad-hoc networks. Users through the `Scenario` class may also decide what kind of mobility pattern (if any) nodes follow, select obstacle and failure models etc.

`Scenario` also handles the deployment and positioning of nodes within the network area. Node deployment can be executed at the beginning of the simulation or *incrementally*

in several phases. Each deployment phase can be run at an arbitrary simulation time and is associated with a positioning scheme; scenarios with incremental node deployment are studied in [8]. `Scenario` implements a simplified positioning scheme used to position the nodes while still respecting the `MotionSpace` constraints. We provide uniform randomized positioning and simplified non-uniform cases where nodes are positioned with different densities according to the network area (such a scenario is described in [7]). Generally, where custom functionality is required, `Scenario` can be extended using the usual TCL mechanism of `instproc` to override the default behavior.

5. Implemented Models

Mobility and Data Collection. Except from the usual randomized mobility patterns such as Random Walk and Random Waypoint Model we also integrate several other algorithms that combine data collection and mobility. These patterns include mobility based on probabilistic criteria and biased probabilistic choices. Specifically, using per node statistics collected by `MobileNodeEx`, we implement mobility models that favor probabilistically less visited areas or areas with high or low node density. We also implement controlled mobility where nodes follow predefined trajectories that are simple geometric shapes; we implement walks on a line, on a circle or ellipse, snake-like, etc. Also, where meaningful, we provide appropriate routing agents that operate jointly with these types of mobility patterns. In [7] such configurations of combined mobility and data collection are described and studied in detail. Also, we implement reactive mobility patterns that can modify the path in response to message reception or other events, such as the Equidistribution protocol which is described in [14]. Note that most of these patterns have been reimplemented in the context of this framework, thus performance may vary from the original version. We note that our toolkit can accommodate additional mobility settings in a flexible manner.

Obstacle Models. Our current implementation uses a number of basic obstacle models defined in [10]. The basic criteria that differentiate obstacles are their *nature* (blocking communications or not), *shape* (circle, square, ring, crescent, rectangle), and *determinism* (probabilistic or not). Deterministic obstacles are always created at the start of simulation time. Probabilistic obstacles can be created at any point in time and can have a limited lifespan. When created, an obstacle is assigned with its basic properties: shape, size, and location. We also make the assumption that all nodes residing inside obstacles are deemed "shut down".

The effect of obstacles on communication inside the network field among "alive" nodes is determined dynamically for each node when it determines its network neighborhood. This is a two-way relationship, since when obstacles change, nodes in the area are notified of the change and recalculate their neighborhood. Currently, our implementation for communication obstacles uses a simplistic scheme for determining whether an obstacle is blocking communication between a pair of nodes: communication is blocked if an obstacle is blocking the line-of-sight. This is of course a simplistic way to determine such properties, and we intend to integrate more realistic models in our framework for this specific purpose, since its architecture allows such extensions. Also, one can introduce "worst case" obstacles, i.e., obstacles which are difficult to efficiently bypass (such obstacles may include mazes and crescent-shaped obstacles with concave parts that can "attract" and trap messages, even when using sophisticated routing protocols. Furthermore, we plan to combine mobility and obstacle presence by studying obstacles that move with time.

Failure Models. Failure injection in wireless network simulations has so far focused mainly on using randomized schemes, namely randomly uniform injection of failures into nodes. We argue that while such schemes are useful, one should also cater for *node-specific* and *location-specific* failure injection schemes. This is due to the fact that in wireless networks in some cases there may be some "key" nodes in the network, regarding their purpose and overall importance. E.g., mobile gateways or cluster heads in WSN are in many ways much more important than ordinary sensor nodes. So, *focused* failure injection under certain circumstances could have far greater impact than a random uniform injection scheme. Also, locating areas of the network field with weaker connectivity, e.g., by examining node density, and focusing on injecting failures on such specific areas could have similar results. Such failure schemes can be easily realized in our framework, because of the efficient implementation of our framework. We plan to include certain types of failures, such as temporary failures, as well as malicious failures relevant to threat attacks. Also, we consider studying a large spectrum of failures strength, e.g., with respect to the failure probability value. Also, location-specific failure injection is supported through evolving obstacles placed in the network field. As obstacles' shape and size may change over time, they cover different parts of the network field. Since obstacles affect all nodes placed inside the region they cover, when they expand to areas that were obstacle-free all nodes in these specific areas are "shut down".

6. Performance Evaluation

We first measure the overhead of our toolkit and more specifically of the mobility domain. Note that in **ns-2** there is a simple walk implementation which is executed online during the simulation time. We implement an identical

`MobilityModel` and compare the two implementations measuring the absolute execution time and the maximum virtual memory usage by the simulator process. Since in both implementations the trajectories are computed dynamically, we are able to measure exactly the overhead of the `MotionDomain`. We conduct several experiments keeping a relatively small network area of $200m \times 200m$. By setting the area small we made sure that nodes will quickly reach their destination and invoke the `MobilityModel` multiple times. We deploy $\{100, 200, ..., 1000\}$ nodes, we set the maximum node speed to $5\frac{m}{s}$ and we simulate 500 seconds of movement without transmitting any messages. We execute 10 times each scenario and average the results. Our experiments were executed on an Intel Pentium M 1733 MHz with 512 MB of RAM, the operating system was GNU/Linux 2.6.20 and the version of **ns-2** was 2.30.

In Fig. 6a we notice that there is a very small overhead,regarding execution time, when using our toolkit, which depends on the number of nodes that are simulated. As the number of nodes increases the overhead tends to becomes larger, however the time complexity is in the same order of magnitude and the overhead never exceeds 25%. Thus, we conclude that there is a small overhead that does not affect the simulation time to a point where conducting large scale simulations is prohibitively slow. When considering the memory usage (see Fig.6b) we can see that the overhead is practically insignificant. Again as the number of node increases the overhead slightly increases but even for 1000 nodes the actual difference is less than 1 MB of memory. Of course the implemented `MobilityModel` holds minimal state and in more complex cases the overhead will increase, but this proves that the framework we implement has a very small memory footprint.

Since our framework extends core components of **ns-2**, it is straightforward to improve even the low-level functionality of the simulator engine. We experimented with the components that calculate the affected nodes on each message transmission as described in Sec.4. By using appropriate spatial data structures in place of the linked list based implementation of **ns-2**, we were able to gain a significant increase in speed. In the following experiments we used a k-d tree but we are in the process of adapting special kinetic data structures. We should note that the impact on memory consumption is negligible (something expected since a k-d tree requires memory linear on the number of nodes, just like the linked list).

Secondly, we measure the speed-ups achieved in the execution of **ns-2** when using the `NodeLocation` component compared to the original implementation of **ns-2**. We performed a set of experiments, using a scenario consisting of a set of wireless nodes exchanging regular TCP traffic (in this case FTP). The number of nodes was in the range $\{100, 200, ..., 1000\}$, deployed in an area of $500m \times 500m$.

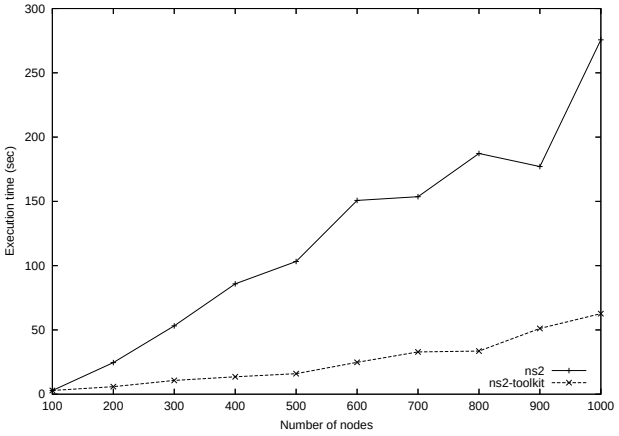


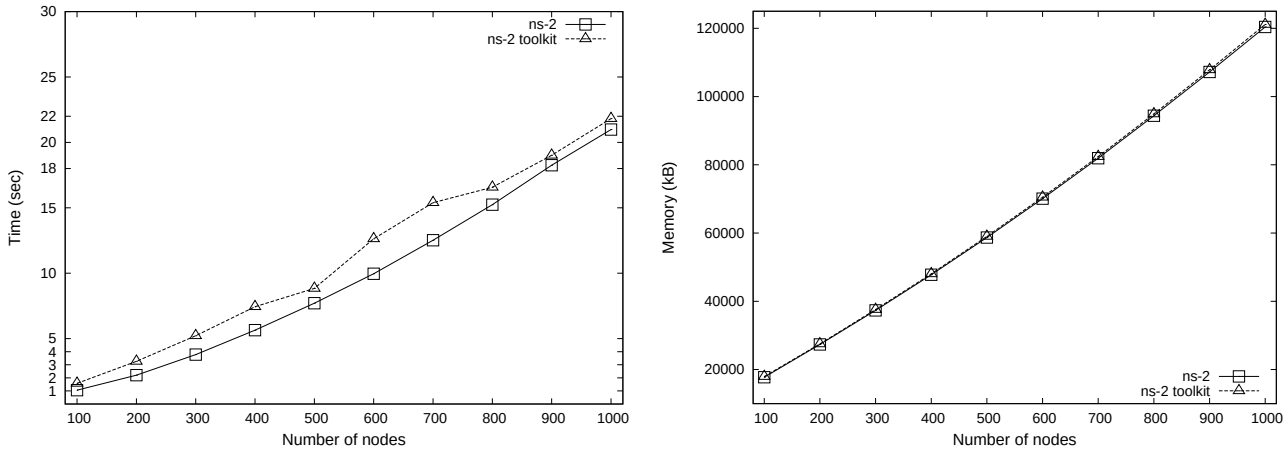
Figure 7. Comparing execution time for original ns-2 and ns-2 with improved node proximity component.

The simulation setting was the same as in the previous set of experiments (OS, CPU, memory). Our results indicate that gains in execution time are quite significant, as can be seen in Fig. 7. In fact these speed-ups are related to the ratio of area to transmission range.

7. Conclusions & Future Work

We presented the design and implementation issues of a novel extension framework for **ns-2**, that especially enables the simulation of dynamic scenarios, particularly in the context of mobile ad hoc, wireless sensor and vehicular ad hoc networks. We evaluated an implementation of our framework and compared it to the **ns-2**, showing that the performance enhancements we introduce for the node proximity component yield a significant improvement on the execution time. When testing our dynamic scenario constructs, we noticed a small overhead on execution time and an insignificant overhead on memory usage. These results, in conjunction to the new functionality and coupled with the ready-to-use library of models and the rest of the enhancements provided by our framework, suggest a powerful tool for modeling and simulating dynamic wireless networks.

Currently, we are in the process of organizing our code in a easy-to-deploy package. This will also include extensive documentation about the architecture and use of our toolkit. More information about the current state of the toolkit can be found in the relevant web page [3]. In the future, we plan to refine the architecture of our framework, improve its genericity and enrich the library of implemented modules with more realistic models. Furthermore, considering that the third version of the Network Simulator is in development, we plan to examine ways of porting our toolkit to the



(a) Execution Time

(b) Memory Usage

Figure 6. Comparing the toolkit overhead on a) execution time, b) memory usage.

new simulator version. Additionally, we plan to study other discrete event simulators, in particular OMNeT++, and examine the possibility of porting our toolkit to another simulation platform.

References

[1] Bonnmotion: A mobility scenario generation and analysis tool. `http://web.informatik.uni-bonn.de/IV/Mitarbeiter/dewaal/BonnMotion/`.

[2] Ns-miracle: Multi-interface cross-layer extension library for the network simulator. `http://www.dei.unipd.it/~maguolof/doc-ns-miracle/index.html`.

[3] TRAILS: Toolkit for realism and adaptivity in large-scale simulations. `http://ru1.cti.gr/TRAILS/`.

[4] Udelmodels, simulation of urban mobile wireless networks. `http://udelmodels.eecis.udel.edu/`.

[5] N. Aschenbruck, M. Frank, P. Martini, and J. Tolle. Human mobility in manet disaster area simulation - a realistic approach. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, 2004.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[7] I. Chatzigiannakis, A. Kinalis, and S. Nikoletseas. Sink mobility protocols for data collection in wireless sensor networks. In *4th ACM/IEEE International Workshop on Mobility Management and Wireless Access Protocols (MobiWac)*, pages 52–59, Torremolinos, Spain, October 2006.

[8] I. Chatzigiannakis, A. Kinalis, and S. Nikoletseas. Adaptive energy management for incremental deployment of heterogeneous wireless sensors. *Theory of Computing Systems (TOCS) Journal*, pages in print to appear in 2007,, 2007. Also, in Proceedings of the Seventeenth (17th) Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2005), ACM Press, 2005, pages 96–105.

[9] I. Chatzigiannakis, A. Kinalis, and S. E. Nikoletseas. Efficient and robust data dissemination using limited extra network knowledge. In *2nd IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 218–233, 2006. Also, in the Journal of Parallel and Distributed Computing (JPDC), accepted, to appear.

[10] I. Chatzigiannakis, G. Mylonas, and S. Nikoletseas. Modeling and evaluation of the effect of obstacles on the performance of wireless sensor networks. In *39th Annual ACM/IEEE Simulation Symposium (ANSS'06)*, pages 50–60. ACM/IEEE, SCS, April 2006. Also, to appear in the Transactions of the Society for Modeling and Simulation Journal.

[11] I. Downard. Simulating sensor networks in ns-2. Technical report, Naval Research Laboratory, 2002.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional Computing Series, 1995.

[13] A. Jardosh, E. Belding-Royer, K. Almeroth, and S. Suri. Real world environment models for mobile ad hoc networks. *IEEE Journal on Special Areas in Communications - Special Issue on Wireless Ad hoc Networks*, 23(3):622–632, 2005.

[14] A. Kinalis and S. Nikoletseas. Scalable data collection protocols for wireless sensor networks with multiple mobile sinks. In *40th Annual ACM/IEEE Simulation Symposium (ANSS'07)*, pages 60–69, March 2007.

[15] S. Kurkowski, T. Camp, and M. Colagrosso. MANET simulation studies: The incredibles. *Mobile Computing and Communications Review*, 9(4):50–61, 2006.

[16] A. Mahajan, N. Potnis, K. Gopalan, and A. Wang. Modeling VANET deployment in urban settings. In *Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*, pages 151–158, 2007.