REIHE INFORMATIK

TR-2005-005

Huginn:

A 3D Visualizer for Wireless ns-2 Traces

Björn Scheuermann**, Holger Füßler*,
Matthias Transier*, Marcel Busse*, Martin Mauve**, and
Wolfgang Effelsberg*

* Universität Mannheim
Praktische Informatik IV
A5, 6
D-68159 Mannheim, Germany

** Universität Düsseldorf
Lehrstuhl für Rechnernetze
Universitätsstraße 1
D-40225 Düsseldorf, Germany

Huginn:

A 3D Visualizer for Wireless ns-2 Traces

Björn Scheuermann[†*], Holger Füßler[†], Matthias Transier[†], Marcel Busse[†],
Martin Mauve[*], Wolfgang Effelsberg[†]
[*]University of Düsseldorf, [†]University of Mannheim

Abstract

Discrete-event network simulation is a major tool for the research and development of mobile ad-hoc networks (MANETs). These simulations are used for debugging, teaching, understanding, and performance-evaluating MANET protocols. For the first three tasks, visualization of the processes occurring in the simulated network is crucial for verification and credibility of the generated results. Working with the popular network simulator ns-2, we have not yet found a visualization toolkit capable of reading native ns-2 trace files and providing means to change the evaluated parameters without changing the visualization software. Thus, we developed *Huginn*, a software providing an intuitive way to visualize simulation properties and to determine how they should be displayed without the need of programming. In addition, Huginn has a 3D interface allowing a high exploitation of the (human) user's perceptive system. It helps to handle the significant cognitive load associated with the mental reconstruction of simulated network processes. Besides presenting the software interface and architecture, we describe algorithmic solutions that might be of a more general interest for similar problems.

I. Introduction

Network simulation is an important tool in mobile ad-hoc network (MANET) research. Usually network simulators create huge sets of 'trace files', i.e., protocols of the discrete events that were processed by the simulator. These files are used for both statistical evaluation of the protocol performance *and* debugging. The former procedure, also call it 'macroscopic evaluation', examines the whole (set of) trace(s) and computes statistical estimators for system properties like, e.g., the 'average ratio of packet delivery'. The latter activity means to look at single events inside a trace file to understand the protocol's (mis-)behavior. Consequently, we call the second process 'microscopic evaluation'. Micro-analyzing a trace file usually challenges a protocol developer with (a) reading

a cryptic trace file and understanding the semantics of a single event, (b) mapping this event to a network node at a certain geographic position, time, and an inner state often not included in the particular event, and (c) including the context defined by other network nodes being close by, also with their respective states. In the past, the complexity of this task sometimes lead to minor and major mistakes that might have been identified and avoided by means of a proper visualization increasing the general credibility of simulation studies [10]. For example, [8] describes a performance optimization of the ns-2 channel. However, this can create situations where nodes (don't) receive transmissions they should (not) receive [5]. This is incredibly hard to find without visualization.

Since finding and interpreting all the relevant information and context within the purely textual data of a trace file is obviously very hard, we have created *Huginn*[1], a visualization tool [14] for packet-level simulation traces. One of the most appealing features of Huginn is its ability to snapshot a two-dimensional MANET scenario using the third dimension to display additional information about both packet transmission processes currently being executed and inner state of the network nodes. This is similar to [11] stating that 3D maximizes the use of the screen and helps to shift the user's cognitive load to the human perceptual system.

The main design constraints were ease-of-use and flexibility to changing user requirements. In its current state of development, Huginn is able to handle both the old and the new wireless trace formats of ns-2 [9], one of the most popular simulation frameworks for network simulation, but could be extended to other formats with justifiable effort. The focus on ns-2, however, creates some problems we will address and we will provide hints on how they could be avoided.

The remainder of this work is structured as follows: While the next section deals with related work, Section III describes Huginn from the perspective of a user. Section IV shows the main aspects of Huginn's software architecture, and the following Section V gives insights into challenging algorithms used to address specific problems in discrete-event visualization. Finally, Section VI concludes the paper and gives a prospect on future work.

II. Related Work

Since the process of micro-analyzing a discrete-event trace is so demanding, assisting it with software tools is not a new idea. In the following we will focus on the major projects available on the web.

[1]Huginn is one of god Odin's ravens in Nordic mythology. The meaning of the word is 'thought'.

Apart from plotting a snapshot of node positions with tools such as *gnuplot* [3], which is the basic approach almost every ns-2 user has once used, ns-2 itself is packaged with *nam*, the *Network AniMator* [2], with the basic purpose of visualizing ns-2 (wired) trace files. To accomplish this, an additional, more expressive type of trace file is used. Nam produces a 2-dimensional display of the wired network with the ability of jumping to an arbitrary moment in simulation time and selecting additional information by using a point-and-click interface. Recent versions of nam also provide support for the visualization of wireless traces. As for now, nam only supports the visualization of statistical values already calculated and stored in the nam trace file. Thus, a major problem with nam's approach is that a new simulation run is needed whenever a different visualization is desired.

The second tool we are having a look at here is *ad-hockey*. It was first created when ns-2 was extended to support wireless networks [7], [1]. Consequently, this Perl/Tk-based tool visualizes mobile nodes and wireless network events based on the standard ns-2 trace format. However, apart from performance problems, ad-hockey is not able to correlate send and receive events.

The third well-known tool around is called *iNSpect* and was developed at the Colorado School of Mines [6]. This more recent project allows to keep lines between mobile wireless nodes, even when transmission events are already over. Thus it is possible to visualize end-to-end routes by keeping lines in place. The main drawback however is the processing of the ns-2 trace file prior to visualization. Moreover, the user has to provide an appropriate parser for this task. Adding up to this, iNSpect does not support jumping to arbitrary time positions.

III. Huginn from a User's Perspective

After the ns-2 simulation runs have produced large amounts of trace file data, a user needs some way to deal with it. This is where Huginn can help and ease the evaluation of the simulation traces.

A. General Functionality

Huginn parses ns-2 wireless traces and visualizes the events having occurred during the simulated time. It displays the simulated scenario in three dimensions with cones depicting the nodes in the network. As the nodes move in the simulation, the cones move around the scene in Huginn. Since ns-2 does not fully support three-dimensional scenarios, the third dimension can be used to display additional statistics in the form of bar charts or text floating above the cones. An example of a 3D view generated by Huginn can be seen in Figure 1. Of course, a static screenshot does not really

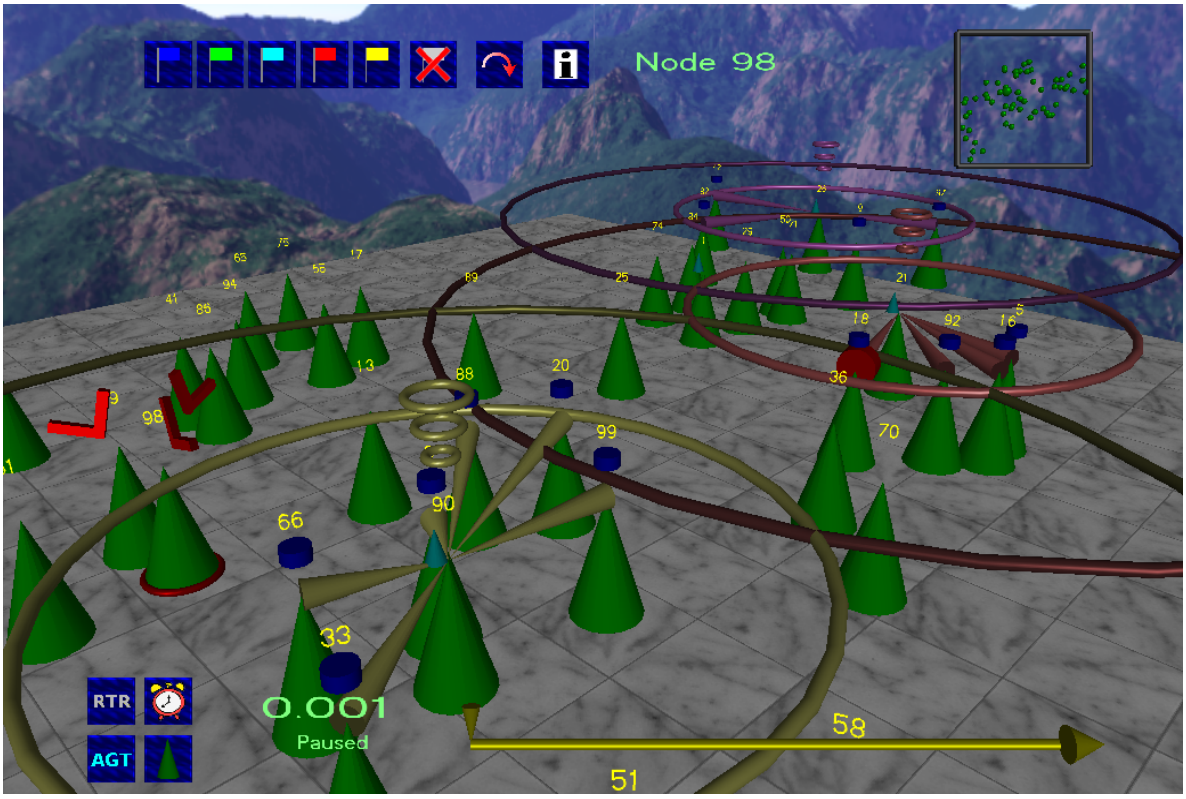

Fig. 1. Example of a 3D view generated by Huginn

give a good impression of how Huginn works. To alleviate this, two little movies showing Huginn in action are available at [4].

The second element which is visualized are the transmissions of packets. When the simulation traces contain MAC layer events, Huginn will scan and combine these to derive information like the sender-receiver pairs of all packet transmissions. Transmissions are depicted by circles around the sender, one circle for the transmission range and another one for the carrier sensing range. This helps the user to see at once where two or more transmissions have blocked each other. In case of unicast, the sending and receiving nodes are—additionally to the range circles—connected to each other by a horizontal cone. If a packet has been dropped at a receiving node because of a collision, this is indicated by a red symbol at that node.

During the visualization, the user has several options to move in time. A time line at the bottom of the screen shows the current position within the simulated time. By clicking on this line, the visualization jumps to the corresponding point in time. Furthermore, there are two options available for the animation of the processes during the simulation: The *linear time* scale and the so-called *FlexTime* scale. The linear time scale has different scaling factors mapping simulation time to

real time, allowing the user to watch the simulation results at different visualization speeds. The *FlexTime* scale adapts the visualization speed to the amount of events happening in a certain period of time. If there is no data traffic for some time and thus no transmissions are to be shown, the visualization speeds up until the next transmission events occur. Then it will slow down again and by this means facilitate the detailed observation of the activities. This mode is particularly useful for simulations with an inhomogeneous distribution of events, for example setups with long periods of inactivity.

While watching the visualization the user may navigate freely through the scene using the mouse or—with even greater degree of freedom—a joystick. This bears two major benefits: first, it allows to look at the scenes from different angles or sides. And second, the user may zoom in and out to take a closer look at a single transmission or small area, or to get an overview of what is happening in the whole network.

B. Adapting the Visualization

After invoking Huginn the user is presented a graphical user interface. It allows to configure the visualization of the simulation data in a simple way. The functionality is arranged into three tabs, the first of which is named 'Nodes': A graphical flow chart, which can be created and edited by the user, determines a pipeline of operations (or a set of different pipelines) through which all lines of the simulation traces will go.

The pipeline consists of five steps: The first step extracts specific events or states out of the stream of trace lines. These events all bear a reference to a point in time and a network node. This first column can be seen as the source of all events, whereas the rightmost column represents perceivable elements influenceable by these events like, e.g., some displayed text. The three columns in between serve as intermediate steps for filtering, aggregation and scaling.

Figure 2 shows an example for such a set of user-defined pipelines. The leftmost column defines the events and states to be monitored. Examples for this are the event of a starting transmission on MAC layer or the current speed of a node. The second column contains filter blocks which can be specified by entering snippets of Ruby [12] code directly within the GUI. An example for this would be that only MAC transmissions of packets larger than 100 byte are to be considered for further processing.

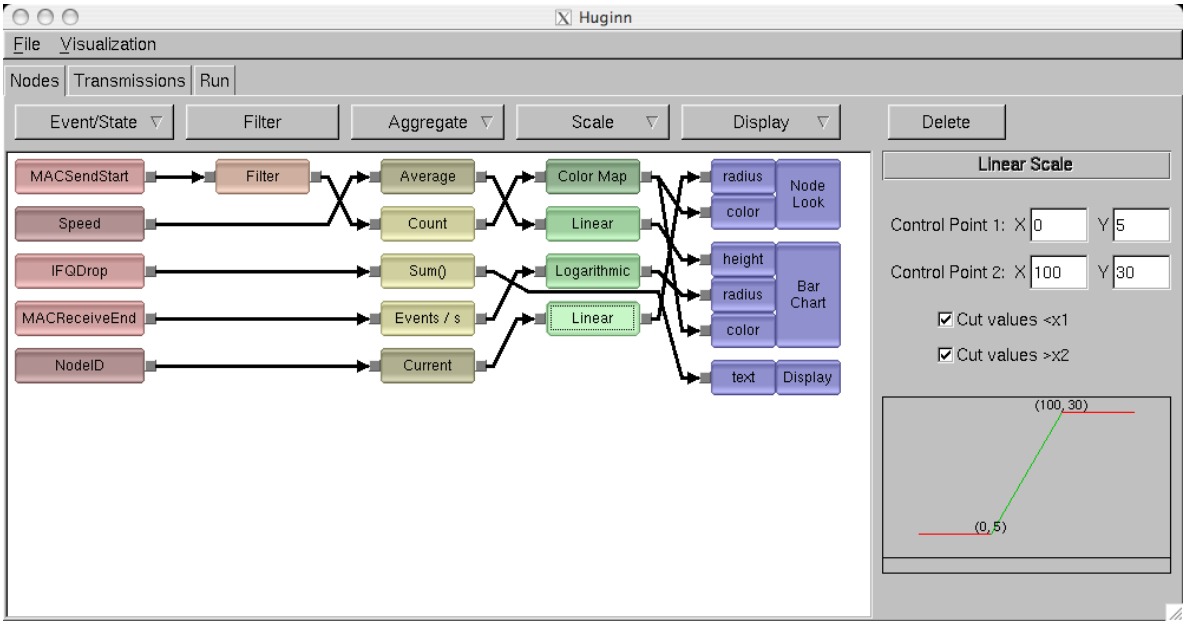

Fig. 2. Example of a configuration in the flowchart editor

After the desired set of events is defined, the middle column specifies aggregate functions. For events like the MAC transmission start, this can be, e.g., the number of events in total. On the other hand, state like the current node speed may for example be averaged in this stage.

While the output values of the third column are already of interest for the user, they are yet to be presented in a perceptive way. To bridge the gap between those values and the perception modules in the right-most column, the fourth column allows to map the values to different scales or colors. An example for the color mapping would be that depending on the number of events, the resulting color slowly changes from green to yellow and finally to red. As a scale a linear or a logarithmic function can be defined to adapt the values to be used in the last step. The interval of the averaged speed, e.g., can be changed from 0 to 20 m/s to values between 0 and 50.

The final stage of the pipeline, represented by the rightmost column of blocks, describes the way the values or colors from the previous stages affect the appearance of the nodes. Values, like the number of MAC transmissions, can be used to vary the radius of a node, to display a bar chart above the node using a corresponding radius or height, or to show a text flowing above the node and displaying the value. In the same way, colors may be assigned either to the bar chart or to the nodes itself.

After the description of the first tab ‘Nodes’, we quickly cover the second, named ‘Transmissions’, and the third one, called ‘Run’. The ‘Transmissions’ tab works similar to the ‘Nodes’ tab but allows

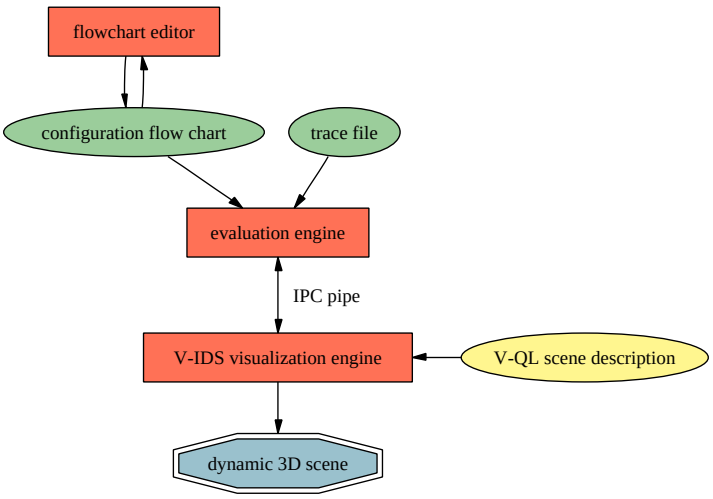

Fig. 3. Huginn software architecture overview

to adjust how the range circles and transmission cones between sender and receiver are represented. The 'Run' tab finally contains the selection of the trace file to be visualized, some other global settings and the button to start the visualization.

IV. Software Architecture

Huginn has three main tasks to perform. It has to

- provide the user with an interface to create and edit configuration flowcharts and set the properties of the flowchart nodes,
- read the trace file and evaluate it according to a given flowchart, and
- render an animated 3D scene and allow the user to interact with it.

The overall architecture of Huginn resembles this structure very closely. The software consists of three main parts, each one responsible for one of the tasks. These three parts are not software components in the classical sense, i.e., linked into one common binary. But they are cooperating very closely at run-time by means of UNIX pipes and configuration files.

The three parts are a flowchart editor for configuring which evaluation is to be done and how the results are to be displayed, an evaluation engine for reading the simulation data and performing the calculations and a visualization engine which transforms the results into a dynamic 3D scene. An overview is given in Figure 3.

A. Flowchart Editor

The first of the three applications is the flowchart editor (FE), which is implemented in C++. This editor is the part of the software the user will see first when starting the visualization tool (see also Section III-B). The FE is responsible for all interaction tasks which have to be performed before starting the visualization with one specific trace file. Also the selection of a trace file to apply the flowchart to and the setting of various global parameters fall into the responsibility of this component.

Additionally, the FE provides means for saving flowcharts into a file and loading them back into the editor. This way flowcharts can be reused later, or can be shared with other Huginn users.

An important aspect of the FE's—and of Huginn's all-over architectural—design is, that the FE does *not* read the trace file. Thus, the configuration of the flowchart, indicating *what* and *how* to evaluate, is completely independent of any specific aspect of the trace file. Any given visualization, once configured using a Huginn flowchart, should be usable with any trace file. This meets the concept of sharing and reusing configuration flowcharts.

When the user has finished configuring the flowchart—or has loaded and potentially modified a saved one—a trace file can be selected and the visualization can be started, combining the data from the trace file with the evaluation rules specified in the flowchart. To accomplish this, the FE starts an instance of the evaluation engine.

B. Evaluation Engine

The evaluation engine (EE) is different from the other parts of the software in terms of its interaction scheme. The EE does not interact with the user at all. It is started by the flowchart editor as a background process when the user wants to apply a flowchart to a specific trace file and watch the resulting visualization. It reads the configuration flowchart transmitted to it by the flowchart editor, and it opens the specified trace file.

During the visualization, the EE reads the trace file and performs the evaluations defined by the flowchart. It manages various data structures needed for the interpretation of the simulation data and for fast navigation within larger trace files. Also, the EE interacts with the visualization engine during a running visualization, providing it with high-level, abstract information about what has to be shown. Reversely, the EE gets information about the user's navigation requests from the visualization engine. For example, the EE is told when the user wants to jump to a given point

in simulation time and then decides, which part(s) of the trace file have to be read and which calculations have to be done to fulfill this request.

The EE is implemented in Ruby [12], an object-oriented scripting language. Ruby is especially suitable for this task, as it is both well-structured and flexible, and—since it is an interpreted language—Ruby programs can easily be extended at run-time. This is used when integrating user-supplied code fragments for advanced evaluation tasks, as mentioned before. These code fragments can be incorporated into the EE by modifying the implementation of an object method within the running software.

Additionally, Ruby's reflective programming features greatly simplify the task of navigating on the time axis while at the same time doing complex (and often irreversible) computations for statistical evaluations. This, along with some other algorithmic aspects of the evaluation engine, will be discussed in some more detail in Section V.

C. Visualization Engine

The visualization engine (VE) used in Huginn originates from another project, V-IDS [15]. It is a flexible, configurable visualization engine for discrete event data. The V-IDS visualization engine can be configured to meet the needs of a specific visualization task by writing a configuration script in a language called V-QL. By providing it with a V-QL configuration script the V-IDS engine has been adapted to the task of visualizing wireless network simulation data.

The configuration script is static and defines the behavior of the visualization engine, i.e., which kinds of objects are or can be present in the visualization, how they fit together, which GUI elements are present in the 3D window, and how the user can navigate and interact with the scene. All Huginn visualizations use the same configuration script, which is not intended to be modified by the user and which is not directly related to the user-generated configuration of evaluations from the flowchart editor.

The visualization engine is a C++ application, OpenGL is used for 3D graphics.

Because the displayed scene might change in each single frame, and because only the evaluation engine reads the trace file and thus can know how and when the scene changes, the EE and the VE have to communicate with each other at run-time. To allow evaluation and visualization to be implemented in two rather lightweight components, the communication between them has to be as small as possible. This is mainly alleviated by three aspects in Huginn's design:

First, the VE has a 'memory' of the visualized scene and allows updates in a differential manner. Only information regarding elements of the scene which have changed from one frame to the next has to be transmitted by the EE.

Second—and maybe even more important: through the V-QL configuration script, the VE has some knowledge on the structure of the data being visualized. This way, the high-level communication between EE and VE mentioned above is facilitated. E.g., when displaying a packet transmission event, the VE simply has to be told that the event exists and which nodes are affected by it; the VE already knows which geometric primitives are to be drawn. From the node position data, the VE can also deduce the position where they have to be drawn, without any help from the EE.

Third, simple interpolation tasks like the linear interpolation of a node's position between two way points can be done by the VE and hence do not cause additional traffic on the communication channel between EE and VE.

V. Algorithms

Huginn visualizes ns-2 wireless network simulation traces as they are written by the simulator. Neither modifications to ns-2 nor any preprocessing of the trace files are necessary.

Reading and analyzing the whole trace file at the start of the visualization is not practicable, since for the usually quite large trace files this would increase both start-up delay and memory consumption. Therefore we have developed algorithms which allow reading the trace file and generating the visualization at the same time.

Although this seems easy to accomplish at first sight, a couple of issues had to be addressed in order to achieve the intended parallel data evaluation and visualization. In general, some of these problems could be circumvented or alleviated by adding information to certain parts of ns-2's traces. We will discuss possible 'improvements' of ns-2 where applicable. For our project, however, we did not want Huginn to require a patched version of ns-2.

A. Look-ahead and Event Queue Reconstruction

In ns-2 a transmission consists of corresponding send, receive, and, possibly, drop events, all represented by a different type of line in the trace file. A send event on MAC layer is logged at the beginning of putting the packet on the channel, whereas receive and drop events (a node receives the packet, but is not able to decode it correctly) occur at the end of the transmission process.

The visualization software has to find all the lines corresponding to one such process to be able to indicate that a packet is not only sent by node *A* but is also received by node *B* or even dropped by node *C*. As a consequence, line-by-line processing is not possible if transmissions are to be shown while they occur. To support the visualization of events when they occur, Huginn has to keep a certain look-ahead, which should be as small as possible, but which should also capture all events related to one radio transmission process.

Further, unrelated simulation events might occur during the transmission and will thus be interleaved into the trace file. Hence the trace file lines describing one single transmission may not form one single, compact block. Even intersections with other MAC layer transmissions—occurring at different locations in the simulation—are very common.

To capture enough lines of the trace file to ensure covering all events possibly corresponding to the current time t_s, the visualization software has to read enough lines in advance to guarantee that all receive or drop lines that belong to any transmission in progress at t_s have been found. This is possible because the temporal extent of MAC layer transmissions has an upper bound. When visualizing t_s, it is sufficient to read the trace file lines with timestamps succeeding t_s until a line has been read with a time stamp greater than $t_s + L$, where L denotes the *look-ahead*, a span of time longer than the longest MAC layer transmission in the trace file.

In Huginn, a fixed (yet user-definable) look-ahead L is chosen before the start of the visualization. The reading of the trace file is always L ahead of the visualization. When the next frame after showing t_s shall visualize the simulation time $t_s + \Delta$, the section $[t_s + L, t_s + L + \Delta]$ of the trace file has to be read before rendering the frame[2].

The information read in advance is needed when displaying any frames between t_s and $t_s + L$. Huginn stores this information in a data structure closely resembling the ns-2 event queue. Events read from the trace file that are relevant for the visualization are inserted into the event queue. In this queue, event connections not explicitly logged by the simulator are reconstructed; send and receive events belonging together are appropriately linked.

For example when after the frame at t_s the next frame is to be displayed at $t_s + \Delta$, the trace file section $[t_s + L, t_s + L + \Delta]$ is read and the events found are inserted into the queue. Afterwards, all events in $[t_s, t_s + \Delta]$ are extracted from the head of the queue. These extracted events determine

[2]Please note that the look-ahead is in *seconds of simulation time* which may result in a varying number of trace lines to read.

all the changes that occur between the frames t_s and $t_s + \Delta$. As described in Section IV-C, a differential update protocol is used for the communication between the evaluation engine and the visualization engine. So the representation in a queue with easy extraction of the changes occurring between two consecutive frames perfectly meets the requirements of the evaluation engine when preparing a frame update.

B. Event Correlation

After having ensured that all potentially relevant trace lines have been scanned, the actual grouping of trace events into a cause-and-effect group has to be done. To correlate send events and their corresponding receive events on the MAC layer, a packet sent by a node *S* has to be recognized when it is received by some other node *R*, i.e., the corresponding send line in the trace file has to be found.

Unfortunately, ns-2 does not supply packet IDs (or something similar) on the MAC layer. In fact, it turned out that it is perfectly possible for two different transmissions to cause completely identical trace file entries.

Due to some assumptions that can be made on the properties of the MAC layer, this problem can mostly be overcome. The central assumption is that when two frames f_1 and f_2 are sent by the same sender, and f_1 is sent before f_2, f_1 can no longer be received after f_2 has been sent. So, only one frame from a given sender can be in transit on the MAC layer at any given point in time. We call this the *single transmission property*. This leads to the idea that when a frame is received, it has to be the last frame sent from the source address in the MAC header of that packet before the reception.

This scheme works well for all kinds of data and routing packets, but fails when used with some special MAC layer frames. In 802.11 CTS and ACK frames the source node's address is not present in the MAC header, in the trace file it is always set to zero. Therefore reliable correlation is not possible using the described method.

It turned out that comparing *all four* fields of the MAC header logged in the trace file entries is a quite good—and the only possible—mechanism to find MAC layer send and receive events belonging together without the need to make modifications to the simulator itself. A reception is associated with the last preceding send event with a completely identical MAC header.

Even this mechanism may theoretically fail, at least for simulation traces using the new ns-2 wireless trace format[3]: If node *K* sends a CTS frame to node *M*, in parallel the node with node ID zero sends an RTS frame to node *M*, *and* the allocation fields in the MAC header (denoting the requested/remaining medium reservation time) are identical, a correct correlation might not be possible. However, we tend to claim that this will not occur very often in practice. A definite solution to this problem would require a modification of the ns-2 trace file format, which is in contradiction of our aim to support the visualization of ns-2 trace files as-is.

On the routing layer a similar solution to the correlation problem is not possible. Since here, too, no packet or event IDs are available and no assumptions similar to the single transmission property of the MAC layer can be made, a reliable event correlation mechanism cannot be established. Therefore, Huginn can only show the occurrence of routing layer events, but cannot decide which events belong together.

On the agent layer the situation is again different. Here, unique packet IDs are indeed present and can be used to find the send event corresponding to a given packet reception. But due to the limited look-ahead, the receptions of a packet will in general not be known when the send event is to be displayed. Since no upper bound for the time between sending and receiving an agent layer packet can be established, extending the look-ahead is not viable. So, Huginn deliberately abstains from any attempt to show agent layer connections, for the sake of simplified and more efficient trace file reading. Nevertheless, for any displayed agent layer receive event the information of the corresponding send event can be accessed, because it must have occurred before the reception and thus must have been read. Then it may be identified by the packet ID. This renders calculations like, e.g., packet transmission delay statistics possible.

To avoid using heuristics, one could of course modify ns-2 to add more information to the trace lines that helps identifying causality relationships. This could even be extended through all layers, to enable the reconstruction and thus the visualization of causality between, e.g., a route request packet and the data packet that caused its sending. However, this would require changes for almost every protocol implementation in ns-2.

[3]In the new trace format, the 802.11 control frame subtype (RTS, CTS,...) is not explicitly logged. In combination with the fact that an unused field is filled with zero, and is thus indistinguishable from a field containing the ID of node zero, this can lead to indistinguishable packets originating from *different* source nodes. Hence the single transmission property is of no avail here. For the old trace format, Huginn is able to use the additional information provided there to resolve this situation correctly.

C. Checkpoint Index

Typically, a trace file visualization will not be viewed linearly from the first to the last simulation second. Instead, certain interesting parts will be repeated, or whole sections might be omitted. In short: It is necessary to allow free navigation on the time axis.

To implement a time axis jump feature it is necessary to provide a means of bringing the visualization software into the right *inner state*. The inner state at time t comprises the structure and values of all data structures, variables or objects as present at time t. In case of Huginn, e.g., the 3D scene as shown at simulation time t, the queue containing the trace file data read in advance (that is, the events in the simulation time interval $[t, t + L]$) and the current value of all statistical evaluations belong to the inner state.

To jump to a given point in simulation time, it is sufficient to construct the appropriate inner state. Of course, this construction has to be possible in a reasonable amount of time. Finding the appropriate position in the trace file is not sufficient, as the information of all preceding events may affect the inner state—just consider statistical values calculated up to this point. Reading the trace file from the beginning up to the right position is a way to construct the desired inner state, but is definitely not fast enough for jump targets which are not too close to the beginning of the simulation.

In Ruby it is possible to make a copy of a complete object hierarchy. Using this feature, it is possible to save the inner state of the evaluation engine into some kind of backup. This backup can be 'restored' later to bring the visualization back to the previously saved state. We call such a backup of the inner state of the evaluation engine a *checkpoint*.

The complete scene can be deduced from the state of the evaluation engine, thus checkpointing the visualization engine is not necessary. Its state can be reconstructed from the evaluation engine's checkpoint.

Checkpoints may contain a considerable amount of data, mostly due to the trace file data read in advance for the look-ahead. Hence the number of checkpoints that can be kept in memory is limited. Most notably it is not possible to create a checkpoint for every single potential jump target, which is every point in simulation time, or at least every point in simulation time a simulation event has caused a trace file entry.

If the inner state at simulation time t shall be reconstructed and a checkpoint representing t is not available, a checkpoint representing a point in simulation time prior to t can be used instead.

After restoring it the section of the trace file filling the 'gap' between the checkpoint's time and t can be processed. Of course, this trace file processing is costly and should be minimized. Therefore it is desirable to have as many checkpoints as possible.

To create checkpoints for the whole trace file, the trace file would need to be processed completely. As mentioned before, this is not an alternative because of the large impact on the start-up time of the visualization. So Huginn creates checkpoints when first reading a specific trace file section. If the user wants to jump into this section again later, the created checkpoints can be used to speed up the navigation.

The cost for reconstructing the inner state for a given point in simulation time grows larger for an increasing number of trace file lines that have to be processed after using the nearest checkpoint before the desired jump target. Thus minimizing the number of these lines is intended. It can be shown that the expected number of lines to be read is minimal for equally distributed checkpoints if equally distributed jump targets are assumed.

Equal distribution of checkpoints cannot be obtained without moving the checkpoints constantly around since the domain they are distributed in is continuously growing while the number of checkpoints has to stay the same. But moving all checkpoints around over and over again would definitely be too expensive. Instead, an algorithm for the creation of checkpoints is used which first creates checkpoints with a high frequency. When the number of processed trace lines and therewith the number of created checkpoints grows, present checkpoints are removed and the frequency of checkpoints is reduced.

The current *checkpoint distance* determines whether after reading trace file line n a checkpoint c_n is created. c_n is created when $n \mod d = 0$. At the beginning the checkpoint distance is initialized to a value of 1000 lines.

The total number of checkpoints is limited by another parameter, the *maximum checkpoint count* m. When the total number of checkpoints reaches this limit, the checkpoint distance d is doubled. This first happens after having read $1000 \cdot m$ lines. New checkpoints are then created with a lower frequency. Each time a new checkpoint is created now, a present one with a position no longer matching the new 'doubled d' is removed. When no more checkpoints exist which can be removed using this criteria, d is again doubled, further decreasing the checkpoint frequency and making half of the existing checkpoints available for substitution.

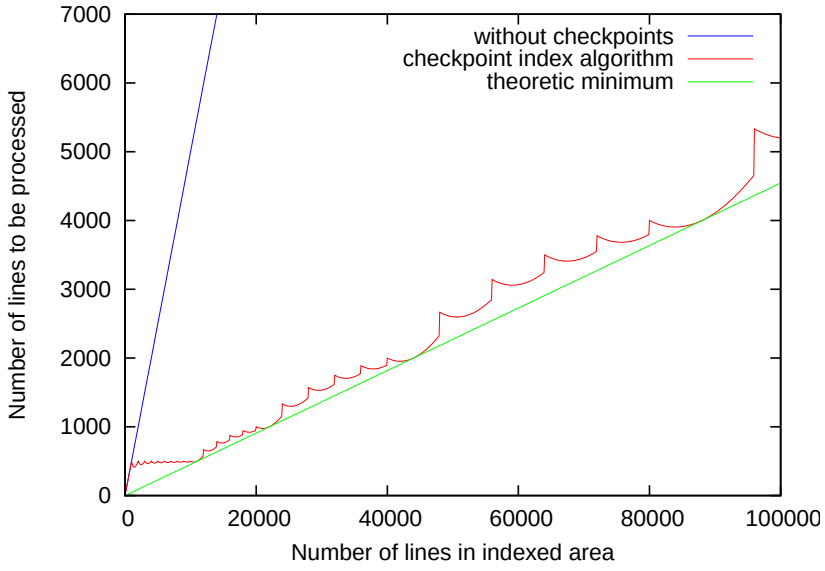

Fig. 4. Expected number of trace file lines to be processed for a jump. The maximum checkpoint count m is 10 and the initial checkpoint distance d is 1000 lines here.

After doubling d old checkpoints no longer matching the new checkpoint distance are not immediately removed. Instead they are kept until being substituted by a new checkpoint. No checkpoint is expunged earlier than necessary, since the information it contains remains equally valuable.

It can be seen that the distance between two contiguous checkpoints created by this scheme is always at least $\frac{d}{2}$ and at most d. A more detailed analysis is provided in Figure 4. We assume an (equally distributed) jump into the indexed part of the trace file and compute the expected number of trace file lines to be processed. The figure shows the results for our checkpoint index algorithm and for the theoretic minimum of at any time equally distributed checkpoints. The performance of the checkpoint index algorithm is close to the optimum, even though in this analytic evaluation a maximum checkpoint count of only 10 checkpoints has been used. In practice, Huginn's default value is 100 checkpoints.

VI. Conclusions and Future Work

Discrete event simulation using the ns-2 network simulator is a very important step in the development and evaluation of mobile ad-hoc networks. Usually, ns-2 writes a textual representation of the events occurring during the simulation into a trace file. Reading this trace file is the main source for understanding MANET algorithms. However, getting a 'bigger picture' than a single event is a demanding task.

Thus, we have developed Huginn, a visualization tool for wireless ns-2 traces. Huginn provides a three-dimensional view of a wireless ns-2 simulation. In addition, it provides an intuitive interface to the visualization of events from the trace file as well as state derived from those events.

To derive wireless transmissions in their geographical and logical scope, algorithms are needed to reconstruct the ns-2 event queue and to find the receive events corresponding to a certain send event. In addition to these, we presented an algorithm for efficient checkpointing.

The main development of Huginn was conducted by the primary author during his master thesis work. Thus, a much more elaborate version of this document can be found in [13]. However, this document is in German language. The home page of Huginn can be found at [4]. On this page screen shots and movies of Huginn in action are provided. In the future, the software distribution will also be available for download there.

While we already have used Huginn for the visualization of our own trace files, there is still some work to do for the easy deployment of the software. As for now, the installation of some necessary support libraries can be difficult. Further, we want to evaluate the usefulness of the Huginn flowchart editor to create macroscopic statistics, providing an understandable tool for the second big part of simulation-driven evaluation studies. In our experience, most people using simulation build their own Perl scripts, which is usually a never-ending source of bugs and non-extendability, given the fact that the metrics evaluated by these kinds of scripts are often quite similar.

REFERENCES

[1] CMU Monarch Project. The CMU Monarch project's ad-hockey visualization tool for ns scenario and trace files. Carnegie Mellon University. http://www.monarch.cs.rice.edu/ftp/monarch/wireless-sim/ad-hockey.ps, August 1998.

[2] D. Estrin, M. Handley, J. Heidemann, S. McCanne, Y. Xu, and H. Yu. Network visualization with the VINT network animator nam. *Technical Report 99-703, University of Southern California*, 1999.

[3] gnuplot homepage. http://www.gnuplot.info.

[4] Huginn homepage. http://www.informatik.uni-mannheim.de/pi4/projects/Huginn.

[5] T. King and T. Butter. Subject: Re: [ns] [bug] Node position not updated. ns-2 users mailing list, October 2004.

[6] S. Kurkowski, T. Camp, and M. Colagrosso. A visualization and animation tool for ns-2 wireless simulations: iNSpect. Technical Report MCS-04-03, The Colorado School of Mines, June 2004.

[7] D. A. Maltz. On demand routing in multi-hop wireless mobile ad hoc networks. PhD thesis, Carnegie Mellon University. http://monarch.cs.cmu.edu/monarch-papers/dmaltz-thesis.pdf, 2001.

[8] V. Naoumov and T. Gross. Simulation of Large Ad Hoc Networks. In *Proc. of ACM MSWiM '03*, pages 50–57, San Diego, California, September 2003.

[9] The ns-2 network simulator. http://www.isi.edu/nsnam/ns/.

[10] K. Pawlikowski, H.-D. J. Jeong, and J.-S. R. Lee. On Credibility of Simulation Studies of Telecommunications Networks. *IEEE Communications Magazine*, 40(1):132–139, January 2002.

[11] G. G. Robertson, J. D. Mackinlay, and S. K. Card. Information visualization using 3D interactive animation. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 461–462. ACM Press, 1991.

[12] The object-oriented scripting language Ruby. http://www.ruby-lang.org.

[13] B. Scheuermann. Dreidimensionale Visualisierung von Simulationsdaten Mobiler Ad-Hoc-Netzwerke. Master's thesis, Department of Mathematics and Computer Science, University of Mannheim, 2004.

[14] L. A. Treinish, J. D. Foley, W. J. Campbell, R. B. Habor, and R. F. Gurwitz. Effective software systems for scientific data visualization. In *ACM SIGGRAPH 89 Panel Proceedings*, pages 111–136. ACM Press, 1989.

[15] V-IDS project. http://www.v-ids.net.