UNIVERSITY OF OSLO
Department of Informatics

3D visualisation of network topology, routing, path distribution and network data in simulated InfiniBand clusters

Master Thesis

Joakim Bjørnstad

May 16, 2011

Acknowledgements

First of all, I would like to thank my supervisor, Sven-Arne Reinemo for his guidance and feedback, in addition to his inspiring and cheerful attitude.

I would also like to thank Simula Research Laboratory for providing a good working environment for the master students. Big thanks to the guys at the lab for great company, many laughs and discussions. A special thanks goes to Bartosz Bogdanski for providing me with the path distribution algorithm.

Last, I would like to thank my friends and family for their support and motivational boosts.

Joakim Bjørnstad, Oslo, 2011-05-16

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and motivation

The demand for more processing power has made us make the move from trying to utilise the processing power of only one computer, to create networks of computers cooperating to solve complex computational problems within a multitude of application areas. These application areas can be everything from physical simulations, monitoring stock markets, to simulating climate changes and earthquakes. The most powerful networked clusters of massively parallel processing computers in existence today are the supercomputers. According to the TOP500 [1] list, which keeps track of existing and planned supercomputers, the number of FLoating point OPerations per Second (FLOPS) a supercomputer is capable of currently is in the order of more than 10^{15} and estimated to break the exaFLOPS (10^{18}) barrier within ten years from now.

To be able to communicate efficiently between processors in a High Performance Computing (HPC) cluster such as the supercomputer, a special type of network have emerged, the interconnection network. The interconnection network is concerned about point-to-point connections directly between processors or to Input/Output (I/O) devices. One of the dominant interconnect technologies in supercomputers today is the InfiniBand Architecture (IBA) [2], being used as interconnect in 42.6% of all the cur-

rent supercomputers in the most recent edition of the TOP500 list [1]. IBA offers a high-performance and scaleable interconnection technology, enabling supercomputers to break new barriers.

With the need for efficient and scalable network solutions for HPC there is a similar need for effective strategies to handle issues around topology, routing, flow control, congestion and deadlocks in such networks. One of the most common tools used by researchers, when developing algorithms and strategies to improve network performance, is simulation. Instead of trying to test for example, new proposed routing algorithms or flow control strategies on real-life InfiniBand clusters, taking up resources from more important calculations, one typically tests them on simulated InfiniBand clusters. This is done by simulating the existence of a complete InfiniBand fabric on a testbed, using dedicated management simulation tools. Data traffic simulation itself, is done by tools such as the Network Simulator 2 (NS-2) [3] or OMNeT++ [4, 5] and the output are *trace files* or *event logs*, containing data about events that happened in the simulation run, for later analysis and evaluation. Trace files can be tedious to analyse manually, using pen and paper. Consequently, researchers write dedicated scripts or programs to answer specific questions about how a given algorithm or strategy performs in a simulation setting. For example, a researcher writing a routing algorithm, wants to see how it performs with a given network topology. Thus, the researcher writes a script to give statistical results on link loads in the topology, output as numbers. In this thesis, we explore the use of a visualisation tool to aid researchers in analysing properties regarding the behaviour of simulated InfiniBand networks. So that the same researcher as in the example, can gain insight about the behaviour and performance of his routing algorithm in a more efficient and time-saving manner, by "*using vision to think*" [6].

For this purpose, several visualisation tools have been created. Examples of such tools for wired scenarios include the Network Animator (NAM) [7] and an option bundled in OMNeT++ [4]. However, these visualisation tools are not able to handle the scale of network topologies commonly present in interconnection networks. Even when the number of nodes reaches the size of a small network topology, the visualisation can become hard to read. This

is because most of the tools mentioned, are concerned about node positions in a network topology using only two dimensions, cluttering up the available display space on the monitor. In addition, they are more focused on giving users the ability to analyse statistical data in topologies and simulations, instead of using visual elements. For example, routing algorithms and network topology together are key when it comes to network performance. Combining these two factors, one can for example create exciting visual maps of the path distribution in a network topology. In turn, such visual maps can lead to a researcher experiencing "eureka" moments, without even having to think, thanks to the advanced human visual system. There also exists general network analysis tools that can handle network topologies with more than a few hundred nodes, such as Gephi [8], Cytoscape [9] and Pajek [10]. But these tools are not able to handle simulation data and are geared toward a multitude of other research fields.

1.2 Problem statement

As stated above, analysing both static and dynamic properties in simulations manually or using statistical tools can be a tedious and a time consuming process. For this reason, visualisation tools have emerged to aid researchers in understanding aspects of network topology, routing, congestion, flow control and protocols. However, not many tools are able to visualise network topologies with nodes in the range of hundreds to thousands in a visually pleasing manner, none are able to visualise the special properties of simulated InfiniBand networks and not many tools have been designed by applying techniques in information visualisation.

Hence, we want to investigate how to make a useful visualisation tool that is able to visualise OpenSM-defined network topologies and routing data in a simulated InfiniBand subnet, in addition to simulated message traffic from OMNeT++. We achieve our results by designing, implementing and evaluating a visualisation tool prototype. The issues we examine in this thesis, are the design challenges when creating such a tool. We want to examine how we can visualise network topology, path distribution, routing and mes-

sage transmissions in simulated InfiniBand clusters. We want to attempt to apply principles and concepts in information visualisation to create a useful and time-saving visualisation tool. We want to investigate how to position nodes in both regular and irregular network topologies, to achieve visually pleasing and readable layouts, using the field of graph drawing. We also want to look into design issues around time complexity and predictability using graph drawing algorithms. We want to explore the use of 3D techniques to help us encode data visually in another dimension. Lastly, we want to take a look at challenges when parsing both static data such as network topology and routing data, in addition to data that changes over time.

1.3 Main contributions

We have presented the design and a visualisation tool prototype called IBSimVis, used to visualise OpenSM-managed network topologies and routing data in 3D. Through visualisation it can aid researchers asking both precise and fuzzy questions regarding a certain network topology or routing algorithm, returning insight on how well paths are distributed in the network. Using IBSimVis for visualising network topologies and routing data, we argue that alot of time can be saved on the researchers behalf. In addition, it can not only help researchers instantly locate patterns or anomalies in a network, but also see the location where in the network topology anomalies occur. Originally we designed IBSimVis to be used for both visualising OpenSM topology and routing data, in addition to simulation message transmissions by parsing event logs from OMNeT++. The visualisation of simulation message transmissions had to be dropped because of time constraints. The reason why we had to do that was because of the time consuming implementation of both 3D and GUI programming, since we had no prior knowledge or experience doing either.

1.4 Language

This thesis is written in UK English. Meaning that some words are spelled differently compared to the their US English counterparts. US English words like “color” and “visualization” are instead spelled in their UK English form, “colour” and “visualisation” [11].

1.5 Outline

In Chapter 2 we present the necessary background material. We take a look at interconnection networks, network topologies and an industry standard for interconnects, InfiniBand. We proceed to introduce the field of information visualisation and look at techniques that can help us achieve more efficient visualisations, in addition to giving an overview of how to draw network topologies. In Chapter 3 we give an overview of visualisation tools that have been created to visualise simulation data for both wired and wireless scenarios. In Chapter 4 we show the design of our visualisation tool prototype and see how it was realised in Chapter 5. In Chapter 6 we evaluate our result, the application we have created and provide a discussion around it. Lastly, in Chapter 7 we conclude and summarise our thesis work and round off by proposing what improvements can be made to the visualisation tool prototype, in the future.

Chapter 2

Background

In this chapter, we introduce the necessary background information for our prototype visualisation tool. In Section 2.1, we introduce interconnection networks, network topologies and an industry-standard for interconnection networks, InfiniBand. In Section 2.2 we introduce concepts and techniques we have used from the field of information visualisation. In Section 2.3, we take a look at challenges and techniques in visualising network topologies, using graph-drawing algorithms. We provide a summary of this chapter in Section 2.4.

2.1 Interconnection networks

An interconnection network, is a switched network, which consists of *nodes*, connected together by *links*. The main purpose of an interconnection network, is to transport data in the form of *messages* between the nodes. Nodes in an interconnection network, can be switches, routers, hosts (also called terminals) or other devices. Interconnection networks that fit the above description, can be found in many different physical scales. They can range from the small, such as the insides of a switch. To the large, such as parallel computer clusters with thousands of processing nodes. These types of networks typically offer high bandwidth and low latency, and are referred to as System Area Networks (SANs). One of the driving factors behind intercon-

nection networks, is the increasing demand in processing power. However, the technology that connects the parallel processing nodes together is a limiting factor [12].

Interconnection networks fall in two categories: *direct interconnection networks* and *indirect interconnection networks* [12]. In a direct interconnection network, every node is both a switch and a host, thus messages are forwarded directly between hosts. In an indirect interconnection network, nodes are either hosts or switches, which means that messages that travel from a source host to a destination host, use an indirect route, passing through one or more dedicated switches.

In the next few sections, we take a look at the structure of interconnection networks (network topology), routing and an industry-standard for interconnection networks (InfiniBand).

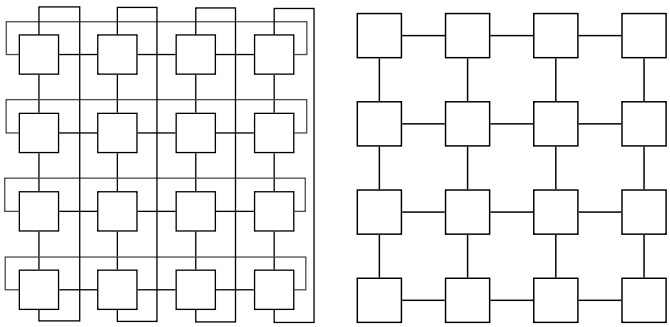
2.1.1 Network topology

A *network topology* defines how nodes and links are interconnected within a network [12, 13]. Each link is bidirectional, meaning each side of a link can both send and receive data. Ideally, each network would be a fully-connected direct network, where each node would have a link to every other node. All messages sent from a source node, would only have to travel to a neighbour, to reach its destination. However, such network topologies are limited by cost, hardware constraints and physical constraints [12]. For example, a fully-connected cabled network requires N ports per node. Depending on the amount of nodes, this would take up alot of physical space in addition to being a costly solution. That is why network topologies that are not fully-connected, have been defined, to reduce the cost and meet the constraints when creating interconnection networks [12].

Network topologies can be divided into two main categories. When we have a topology, that has rules on how switches are interconnected, it is called a *regular topology* [13]. Such rules may involve all nodes having the same degree, or nodes arranged in orthogonal n-dimensional space [13]. When we on the other hand, have a topology where no such rules for the connection pat-

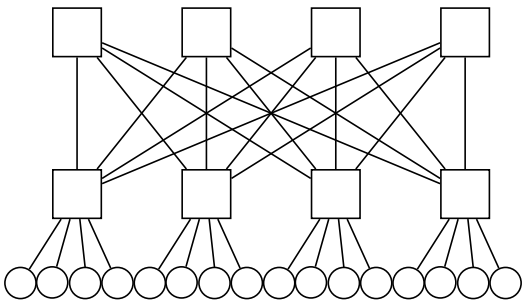
tern is defined, we call it an *irregular topology* [13]. Routing algorithms may exploit the properties of regular topologies. This is not the case for irregular topologies, where general solutions are required [12]. In this thesis, we are interested in being able to visualise both regular and irregular topologies.

Network topologies are typically modelled as graphs, since they share many of the same characteristics [12, 13]. Vertices and edges in a graph, map to nodes and links in a network, respectively. Later in this chapter, we see how we can draw graphs- and thus also network topologies. Next, we briefly introduce and show some common regular network topologies found in interconnection networks. Namely, the torus (k-ary n-cube), mesh (k-ary n-mesh) and fat-tree (k-ary n-tree) topologies. These are part of a parametric family of network topologies, constructed using the parameters k and n, as shown in Figure 2.1.



(a) Torus topology: 4-ary 2-cube

(b) Mesh topology: 4-ary 2-mesh

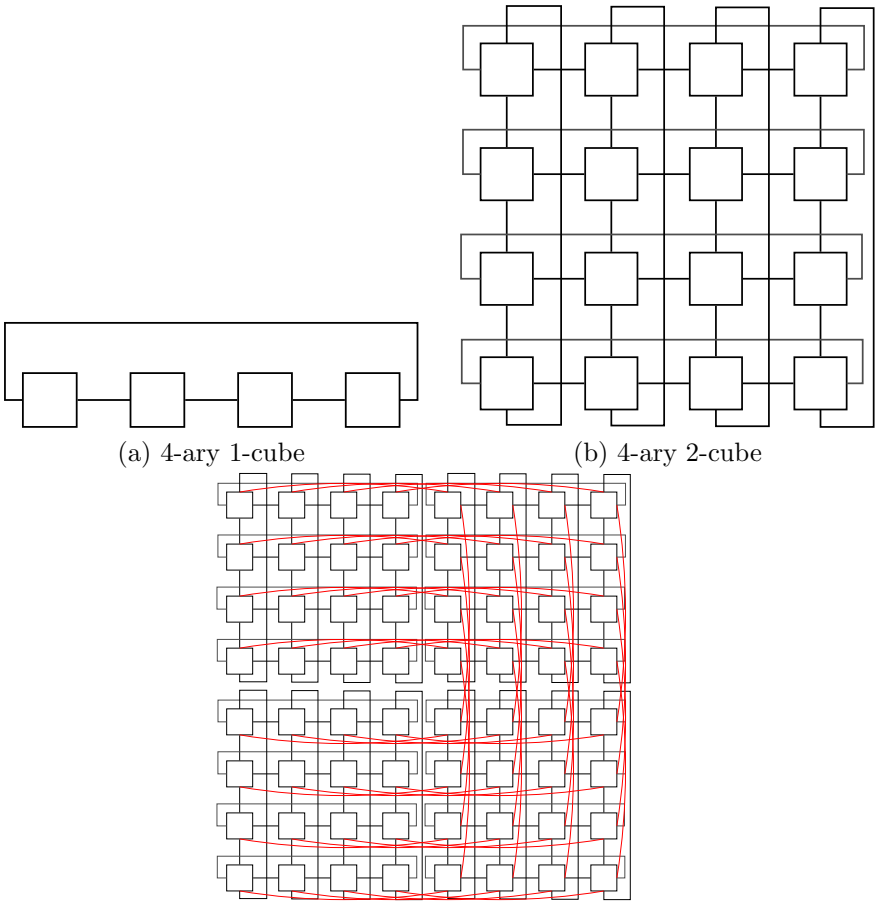


(c) Fat-tree topology: 4-ary 2-tree

Figure 2.1: Three common regular network topologies for interconnection networks: torus, mesh and fat-tree. Using k = 4 and n = 2.

2.1.1.1 Torus (k-ary n-cube)

A torus topology, is a direct interconnection network, constructed as an n-dimensional grid with k nodes in each dimension [12]. The number of nodes is k^n. Figure 2.2 shows how a torus topology grows when the dimensionality, n goes from 1 to 3.



(a) 4-ary 1-cube

(b) 4-ary 2-cube

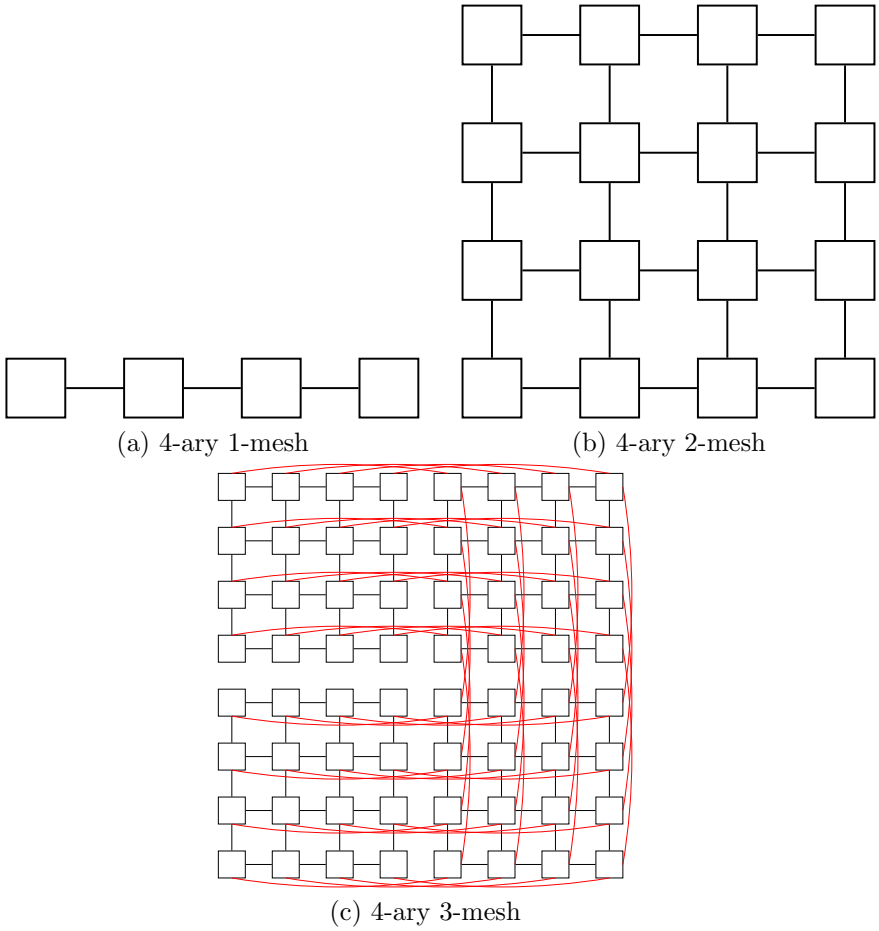
(c) 4-ary 3-cube

Figure 2.2: Three torus topologies, showing how the topology grows when n increases.

2.1.1.2 Mesh (k-ary n-mesh)

A mesh topology, is a direct interconnection network and shares many of the same characteristics of a torus network. The exception is that the connections

from address a_{k-1} to a_0 has been removed [12]. The number of nodes is k^n. Figure 2.3 shows how a mesh topology grows when the dimensionality, n goes from 1 to 3.



(a) 4-ary 1-mesh (b) 4-ary 2-mesh

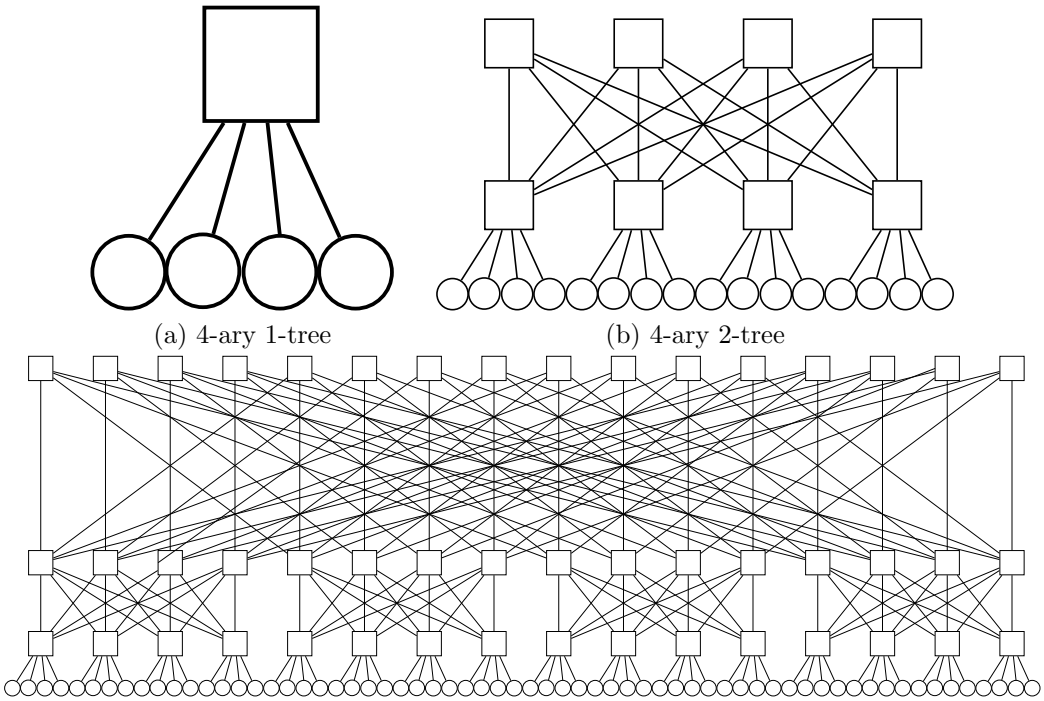
(c) 4-ary 3-mesh

Figure 2.3: Three mesh topologies, showing how the topology grows when n increases.

2.1.1.3 Fat-tree (k-ary n-tree)

A fat-tree topology is an indirect interconnection network [14]. The fat-trees are based on complete binary trees and are named such since they resemble real trees, since they get thicker closer to the root [14]. A special subclass of fat-trees is the k-ary n-tree, having k^n processing nodes and nk^{n-1} switches, arranged in a $k * k$ connection pattern [14]. Fat-tree topologies such as the

k-ary n-tree is commonly found in deployed InfiniBand networks [15]. Thus, they are the most relevant to visualise in our tool. Figure 2.4 shows a k-ary n-tree with the dimensionality n from 1 to 3.



(a) 4-ary 1-tree

(b) 4-ary 2-tree

(c) 4-ary 3-tree

Figure 2.4: Three fat tree topologies, showing how the topology grows when n increases.

2.1.2 Routing

Routing determines the path taken by data messages through a network, from a source node to a destination node. Some topologies offer only one path, such as the one-dimensional regular topologies we have seen. However, in the case of topologies that offer multiple paths between two switches, routing algorithms determine which path a message should take. A routing algorithm is a key factor in network performance, the other being flow control [12]. It is the routing algorithms task, to distribute paths evenly in a topology, so that some links are not more heavily loaded than others [12]. In addition to

keeping the number of *hops*, i.e. the path length as short as possible. We refer to the load or distribution of paths over a network topology as *path distribution*. Note that the performance and characteristics of routing algorithms is not the focus of this thesis. We are merely interested in visualising the effect a routing algorithm has on a network topology.

2.1.3 InfiniBand

The InfiniBand Architecture (IBA) [2], is an industry-standard architecture for inter-server communication, in Local and System Area Networks [16]. IBA has been specified by the InfiniBand Trade Association (IBTA), a group of technology companies, whose Steering Comittee consist of IBM, Intel, Mellanox, Oracle, Qlogic, SFW and Voltaire [17]. Furthermore, the OpenFabrics Alliance (OFA), an open-source community for IBA actors, develops and maintains the OpenFabrics Enterprise Distribution (OFED), a software package used by IBA compatible network clusters [18].

IBA has two main characteristics. First, IBA is using point-to-point connections instead of busses; avoiding arbitration issues, increasing reliability and allowing scaling [16]. Second, transferring data and commands are sent as messages and not memory operations [16]. As illustrated in Figure 2.5, a *subnet* is the smallest complete IBA unit. A subnet is also referred to as a *fabric* or *cluster*. It consists of endnodes, switches, links and a Subnet Manager (SM). Endnodes send messages over links, via switches to other endnodes. Channel Adapters (CAs), connect endnodes to links. Each link is connected to a port on either a switch or a CA. In the next paragraphs, we give a brief description of each element in an IBA subnet, as presented in [16] and selected topics in [2].

Links

IBA supports bi-directional communication, through both copper cable and optical fibre. Physical links can be used in parallel to achieve greater bandwidth. Link widths are referred to as 1X, 4X and 12X. Links are split into logical channels, called Virtual Lanes (VLs). Each link can have a config-

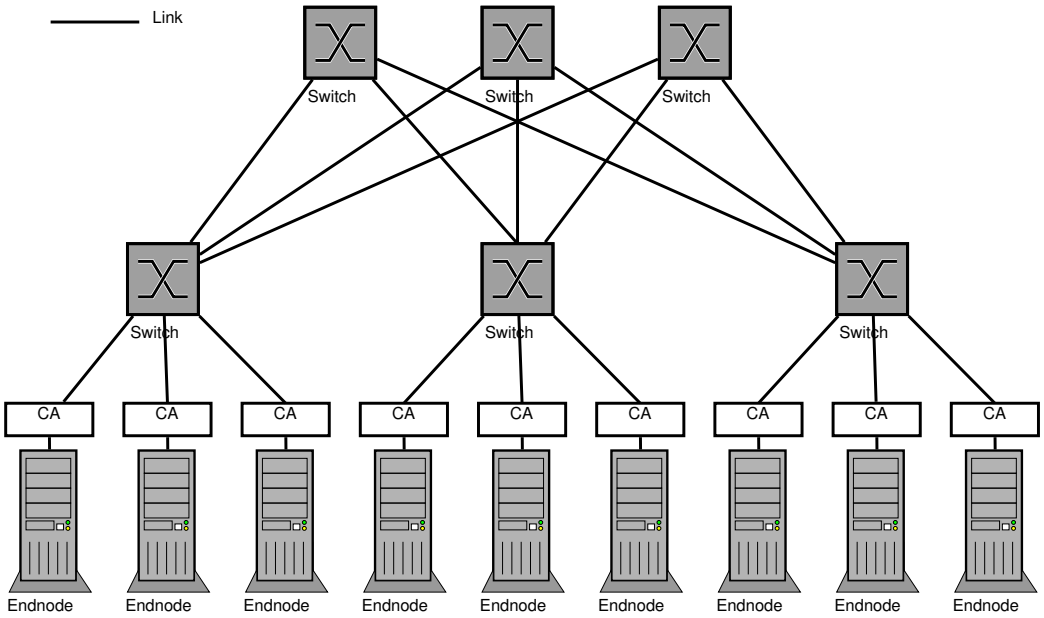


Figure 2.5: An InfiniBand subnet with a fat-tree topology (3-ary 2-tree).

urable number of VLs, up to a maximum of 16. The minimum number of required VLs is two: VL 0 is designated for normal data traffic and VL 15 is designated for subnet management traffic. In addition, each VL has a set of dedicated send- and receive- buffers at both ends of the physical link. Although IBA is specified with support up to 16 VLs, only 8 data VLs in addition to the dedicated VL for subnet management traffic, have currently been implemented in hardware.

Switches

Switches forward messages from source endnode to destination endnode. Each switch has a forwarding table, which is computed during network initialisation and modification. As an example, a forwarding table might be linear, specifying *destination Local Identifier (LID) : outgoing portnumber* pairs, for each possible destination LID in the subnet, as is the case for the OpenSM unicast forwarding file. Both LID address and SM are described below. The amount of ports on a switch is vendor-specific, but the maximum number of ports is 256. Switches have three methods of service differentiation. VL

to Service Level (SL) mapping, weighted round-robin VL arbitration and classification of VL as either High or Low priority.

Endnodes

Endnodes are the packet producers and consumers in an IBA unit. They can be a wide variety of hosts or devices. Hosts can be computing units, while devices can be storage units. Endnodes are connected to CAs, described in the next paragraph.

Channel Adapters

The Channel Adapter (CA), is the interface between a physical link and an endnode. There are two types of Channel Adapters: Host Channel Adapter (HCA) and Target Channel Adapter (TCA). HCAs has a software interface, called verbs and are connected to hosts. TCAs on the other hand, are connected to devices and have no verbs. The main tasks of a CA is to transmit data to endnodes, using one of the communication service types, and segmentation and reassembly of messages. There are five communication service types offered by the CA [2]:

Reliable connection An acknowledged connection between two endnodes, ensuring that data has reached the destination.

Unreliable connection An unacknowledged connection between two endnodes, where lost or corrupted messages are dropped.

Reliable datagram An acknowledged datagram that can be sent reliably to an endnode without a connection.

Unreliable datagram A connectionless and unacknowledged service. Used by the management system to discover the fabric.

Raw datagram (IPv6 datagram and EtherType) A data link service to allow bridging of native Ethernet, TCP, UDP and IPv4 packets into an IBA subnet.

Furthermore, for the CA to communicate using the mentioned communication service types, it utilises work queues called Queue Pairs (QPs). There are three types of QPs: send, receive and completion.

Subnet Manager

An IBA subnet requires a Subnet Manager (SM), residing on either an endnode or a switch. Their main purpose is to initialise and discover the network, assign LIDs to switches and CAs and load the forwarding tables for switches for endnode-to-endnode routing. The SM also provides paths for networks with different MTU. All management information is communicated via special messages, called Management Datagrams (MADs). An implementation of SM software, OpenSM is described in Section 2.1.3.1.

Addressing

Local Identifiers (LIDs) are 16-bit identifiers, used by switches for routing purposes. LIDs are assigned to each switch and each port on an endnode. Note that switch ports do not have an assigned LID address. Global Unique Identifiers (GUIDs) are 64-bit EUI-64 IEEE identifiers (Ethernet MAC address) for elements in a subnet, they are assigned to chassis, CA, CA ports, switch and router ports.

2.1.3.1 OpenSM

OpenSM is an InfiniBand compliant Subnet Manager [19], maintained by the OFA [18] and is a part of the OFED. A software entity such as OpenSM, residing on a node in an InfiniBand subnet, is required in order to initialise InfiniBand hardware [19]. OpenSM attaches itself to a port on an InfiniBand node. It then proceeds to initialise the InfiniBand fabric, by first discovering the fabric using MADs, then by assigning LIDs to switches and CA ports [19].

OpenSM generates a Unicast Linear Forwarding Table (LFT), based on one of several routing engines [19]. The LFT is a forwarding table that exists in every switch. It specifies which port packets should be forwarded through, for every LID in the subnet. Reassigning LIDs requires recomputation and

repropagation of LFTs [19]. Routing engines in OpenSM, are algorithms that compute LFTs, based on the network topology of a fabric. Depending on the OpenSM version, there can be several routing engines present. A few examples of routing engines [19]:

- Min Hop Algorithm
- UPDN Unicast routing algorithm
- Fat Tree Unicast routing algorithm
- LASH unicast routing algorithm

In addition, the user can create his own routing engines.

2.1.3.2 IBMgtSim

The InfiniBand Management Simulator (IBMgtSim) is a management simulator for InfiniBand networks. It simulates the existence of a complete InfiniBand subnet with the number of nodes and topology input by the user. After IBMgtSim has been initialised, OpenSM is able to "discover" the simulated fabric. It is important to note that IBMgtSim only simulates nodes and properties related to nodes in an InfiniBand network, not the actual data traffic. For this, OMNeT++ is used. In this thesis we sometimes refer to the topology and routing data files in OpenSM format. By this we mean that OpenSM discovered and initialised the InfiniBand fabric and dumped the static information to files. While IBMgtSim created the fabric by simulating the existence of all the switches, CAs, ports and links.

2.2 Information visualisation

In the field of visualisation, there are at least two related and overlapping subfields. One, scientific visualisation, is concerned about visualising properties of the real world, usually in 3D, such as medical imagery, mechanical stress and weather simulations [20]. The other, information visualisation,

is about visualising abstract data and concepts, such as networks, finance, statistics [20]. Both fields overlap each other, in the sense that they both use some of the same techniques for visualisation. In this thesis, we are interested in visualising network topologies and related properties in InfiniBand networks, over time in a simulation setting. In Section 2.1.1, we observed that topologies are modelled as graphs, which are collections of abstract data. Thus, techniques in the field of information visualisation is most relevant to apply when we design and implement our visualisation tool prototype.

Card defines *information visualisation* as:

> *The use of computer-supported, interactive, visual representations of abstract data to amplify cognition.* [6], page 7.

Cognition is the process of acquiring and using knowledge [6]. One might ask, how does visualisation amplify cognition? Card [6] and Ware [21] state several reasons:

First of all, our highly developed visual system, gives us the ability to recognize and process patterns efficiently [6, 21]. Second, our visual system allows high bandwidth access from the computer display, to our brain [21]. Third, visualisations expand our working memory for solving problems and can store large amounts of information in a quickly accessible form [6].

Spence notes three primary reasons [20], for why computer-support have aided cognition. Computerized storage mediums with low access times, allowing storage and fast access to large amounts of data. Increase in computer processing power, enabling dynamic selection and interaction with datasets. High-resolution computer hardware and monitors, which is able to present data well enough to match the human visual system. In essence, information visualisation is a set of graphical techniques and methods for gaining insight, understanding and information from *abstract data*.

One of the first examples of information visualisation is the famous image of the Russian Campaign in 1812, during the Napoleonic Wars [20]. Made by Joseph Minard and published in 1869. It shows the advance and retreat by the forces of Napoleon I. As shown in Figure 2.6, line thickness indicates troop numbers and the bottom graph, temperature. The image is a prime

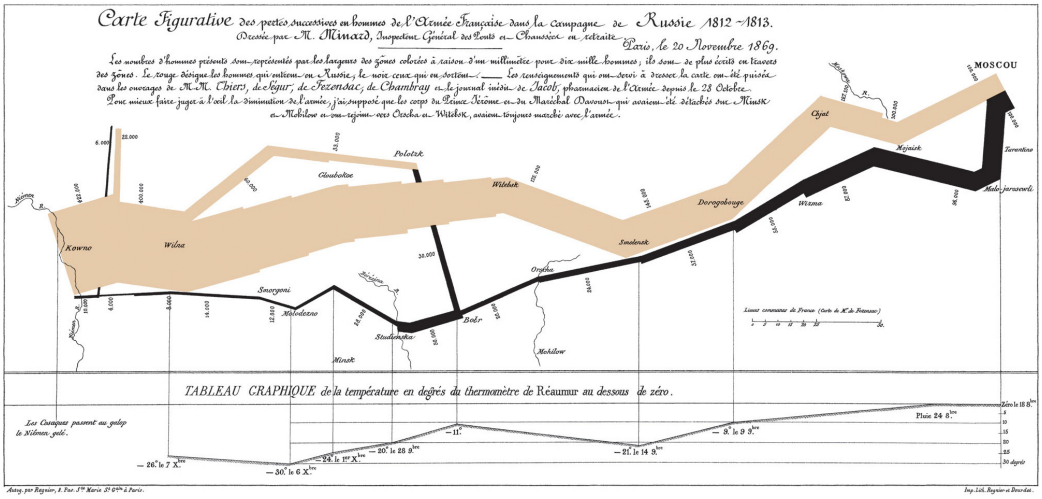


Figure 2.6: Charles Minard's 1869 chart showing the losses in men, their movements, and the temperature of Napoleon's 1812 Russian campaign. Lithograph, 62 x 30 cm. Public domain. [22]

example of how visualisation can aid in the understanding of data.

The initial definition of information visualisation also covers the three principal issues [20]. Those of, how to *represent* abstract data, encode data to visual objects. How to *present* the visual representations to the user, so that the user can interpret the image to gain insight and knowledge. And lastly, how to able the user to view the data from different angles, by *interacting* with the presentation. These issues are covered next and the process is shown in Figure 2.7.

2.2.1 Representation

The first step in the process of information visualisation is to take raw data and find a way to represent them [6,20,21]. Raw data usually takes the form of numbers or strings of text with some semantic. In Figure 2.6, we saw how numerical data such as number of troops was represented as line thickness and the use of colour encoding shows the advance and retreat of Napoleon I's troops. This process of transforming raw data to visual objects is called *encoding* [20]. Before we start encoding data, we need to know what type of data we are dealing with and their complexity [20]. Once that has been

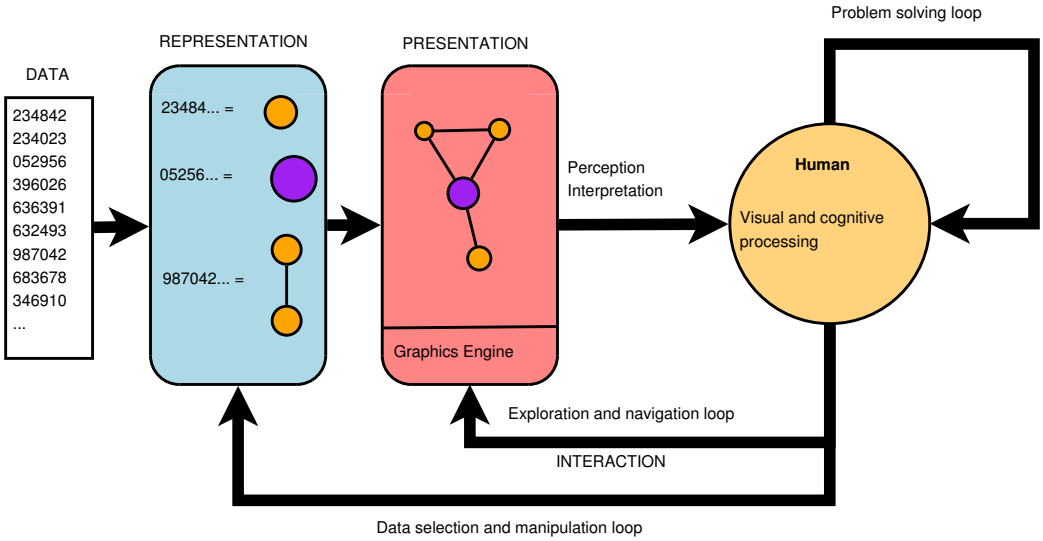
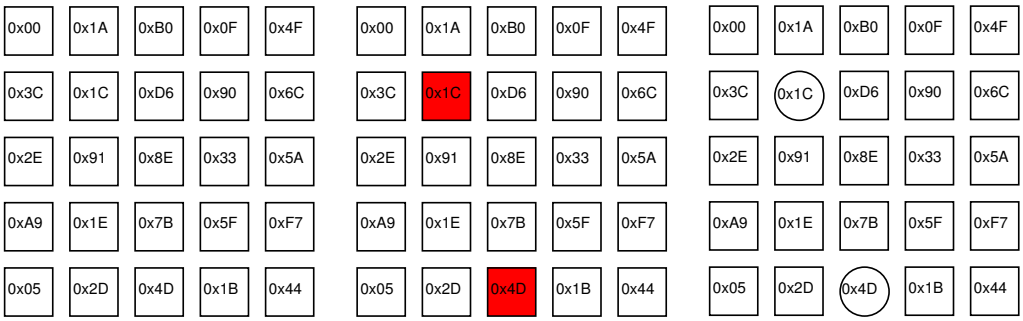


Figure 2.7: The process of information visualisation. Showing the three main issues in information visualisation, in addition to the interaction feedback loops, described in Section 2.2.3. Adapted from Figure 2.17 in [20] and Figure 1.2 in [21]

established, we need to choose which *encoding mechanisms* to apply [20].

There are two categories of data types, *entities* and *relations* [21] (Spence refers to these as value and structure, respectively [20]). An entity is the object we want to visualise, such as a car or a house. Relations are the structures that connect entities together and can be one of two types, conceptual or structural [21]. As an example, the relationship between a car and its owner is conceptual and the relationship between inner components of a car such as engine and tires are structural. Furthermore, entities and relations contain zero or more *attributes* [21]. An attribute is a property that can not be described independently, such as the colour of a car [21]. Attributes of entities can have several dimensions [6,20,21]. *Scalar* attributes is an one-dimensional quantity, such as the number of wheels of a car. *Vector* attributes are two-dimensional, for example the direction a car is travelling. *Tensor* attributes are multi-dimensional and can be used to denote both the direction and force of a car.

Above, we have presented what types of data that exists. Once data has been abstracted into entities, relations and associated attributes, we need



(a) No pre-attentive features.

(b) Coloured.

(c) Different shapes.

Figure 2.8: Example of pre-attentive processing [21]. Try identifying switches labelled 0x1C and 0x4D.

to know how to create their visual representations. We refer to encoded visual representations as *glyphs* or *symbols* [21]. We get the glyphs by applying encoding mechanisms to the entities we want to represent. Here, we come back to what we mentioned earlier about human visual pattern processing. Certain glyphs stand out to our visual system [21]. This is called pre-attentive processing and is shown in Figure 2.8. Visualisations can help us identify patterns or anomalies in data, such as graph structures, by identifying similarities or differences in groups of symbols [21]. Ware notes, that it is desirable for symbols in a visualisation to be preattentively distinct from eachother [21]. Thus, the encoding mechanisms are tightly connected to what visual properties they exhibit.

One set of encoding mechanisms often referred to in literature, are those introduced by the cartographer Jacques Bertin (1918-2010), who is considered one of the pioneers in information visualisation [6, 20, 21]. Bertin introduced the six encoding mechanisms (called "retinal variables" [23]) in graphical representations, that could be applied to entities [6]. The encoding mechanisms introduced by Bertin [23] are shown in Figure 2.11.

The encoding mechanisms in the figure, has later been formalised and evaluated for efficiency and accuracy, by Cleveland and McGill [24] and later by Mackinlay [25]. The encoding mechanisms presented by the authors mentioned above, still keep the basic encoding mechanisms presented by Bertin,

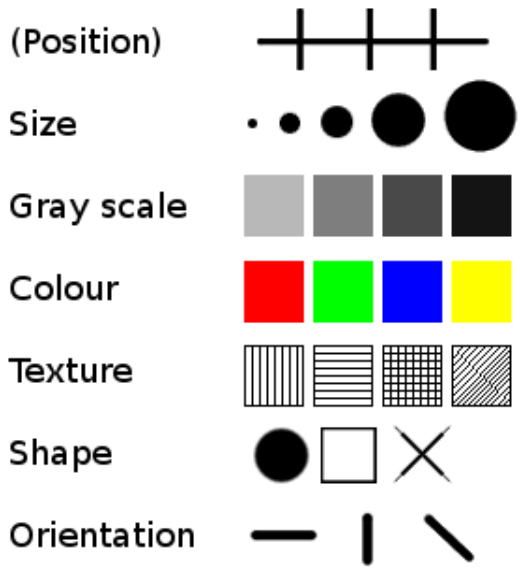
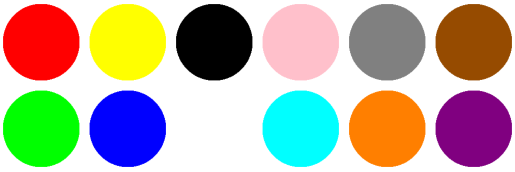


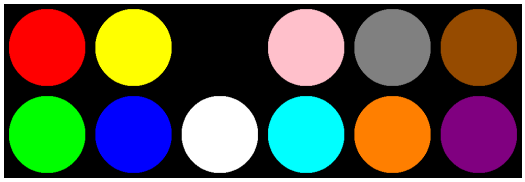
Figure 2.9: Bertin's encoding mechanisms or "retinal properties" [23]. Adapted from Table 1.21 in [6].

with the exception of seperating gray scales from the colour encoding mechanism. With the full list of possible encoding mechanisms being far to many to use in our visualisation tool prototype, we have decided to use the basic ones introduced by Bertin, except the grayscale one, which we incorporate into colour. The reason for this choice is that not all encoding mechanisms are relevant to visualising network topologies. For an overview of the most common encoding mechanisms, we refer to Figure 3.44 and 3.45 in [20]. Regarding the colour encoding mechanism, Ware suggests a set of 12 default colours to use when labelling data. These colours are red, green, yellow, blue, black, white, pink, cyan, gray, orange, brown and purple. They are shown in Figure 2.10, on white and black backgrounds.

We have covered what data types that data belongs to and what encoding mechanisms that are relevant to apply to entities and their associated attributes. However, we have not yet mentioned how to represent relational data, in the form of graphs. Most commonly this is done by constructing



(a) On white background.



(b) On black background.

Figure 2.10: Ware's suggested set of 12 colours used for labelling. Adapted from Figure 4.21 in [21]

node-link diagrams [20, 21]. With nodes representing entities, and with links going between them, indicating relationships. Furthermore, nodes can be encoded according to the attributes that belongs to it or associated data values. They can for example be encoded as specific coloured shapes of specific sizes, depending on context. Links can in turn vary in width and colour.

2.2.2 Presentation

The second step in information visualisation process is to decide how to *display* the representations to the user [20]. Or even more important, if we are to display them at all [20]. Sometimes, there is too much information available at the same time, so there might be a need to hide some information. Either from the start or through interaction. Furthermore, when we have decided what information to display, we have to consider if the user is allowed to interact with the presentation [20].

The most important problem during presentation is the lack of display space [20], the available area of a computer display. Even high-resolution computer monitors can not show all information, at once. Therefore there is a need for techniques that effectively utilises the available display space. The most common techniques is by adding the ability to *zoom* and *pan*. Zooming is the "smooth and continuously increasing magnification of a decreasing

fraction (or vice versa) of a two-dimensional image ounder the constraint of a viewing frame of constant size" [20], page 117. Panning is the "smooth, continuous movement of a viewing frame over a two-dimensional image of greater size" [20], page 117. Zooming can be divided into two activities. *Zooming in* gives the ability of viewing information in a more manageable manner. While the other activity, *zooming out* can give context to information. There are also two types of zoom. *Geometric zooming* or *spatial zooming* allows the user to take a closer look on information, while keeping representations static [6, 20]. For example, when taking a closer look on a map. *Semantic zoom* [26] offers new views on representations depending on zoom distance, making presentations dynamic. For example by zooming in on a city on a geographical map, it gives more information about that city, such as demographics. In Section 2.2.1, we learned that graphs can be represented as node-link diagrams. To address the presentation problem of graphs and network structures, we have dedicated Section 2.3 to the specialised field of graph drawing.

2.2.3 Interaction

> "A good visualization is not like a static picture or a 3D environment we can walk through and inspect like a museum of statues." [21], page 317.

Interaction enables the user to find information in the presentation and glyphs on the display screen, typically using input devices such as a mouse and a keyboard. Interaction is the real strength of information visualisation, giving the ability to change views when looking at data. Interaction is done by the user in three feedback loops. The *data selection and manipulation loop*, where the user selects, filters and manipulates the glyphs [21]. The *exploration and navigation loop*, where the user navigates through the visualisation and identify structural landmarks [21]. Lastly, *the problem solving loop*, where users form questions or gain insight to the visualisation [21]. The feedback loops were illustrated earlier, in Figure 2.7. Next, we briefly introduce the three main interaction techniques we use in this thesis, besides

the already introduced pan and zoom, which are part of the exploration and navigation loop.

Selection and hover queries

The selection query is initiated when the user clicks on a glyph in a visualisation and is one of the most common interactive operations [21]. It is used to retrieve additional entity data.

Colour selection using a colour pallette

A colour pallette enables the users to manipulate entity representations, by changing its colour. For example, the default colour of a glyph, might have been unsuitable in combination with the background or other glyphs in a visualisation. Requiring a method of being able to select different colours for a glyph. One method is to allow the user to type in RGB coordinates in the colour cube, drag on sliders to adjust red, green and blue or choose colours from a pallette [21]. The two first methods might be confusing for the user, since they may not necessarily know much colour theory and do not know what colours to mix in what quantities to get the one they want [21]. Ware states that a colour pallette is a "good solution to the colour selection problem." [21], page 123.

Dynamic Query

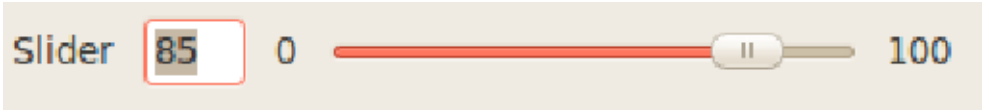


Figure 2.11: Dynamic query in a slidger GUI widget. Adapted from Figure 1 in [27]

A Dynamic Query is an interaction technique to filter data, using graphical widgets such as sliders [27]. Originally the dynamic query was used as an alternative to high-level query languages such as Structured Query Language (SQL), to dynamically fetch data from databases [27]. However, it

can be applied to most scenarios where a user either needs to fetch data or filter data out. The advantages is that limits of the query is displayed graphically, the query result is immediately displayed, it is easy to use even for novices and it is easy to implement [27]. This technique is used in our visualisation tool prototype, when we want to filter out entities when visualising path distribution.

2.3 Graph drawing

> "A good layout can be a picture worth a thousand words; a poor layout can confuse or mislead" [28], page 435.

Earlier, in Section 2.1.1 we learned that network topologies are graphs. And in Section 2.2.1, we learned that relational datatypes, such as graphs can be drawn using node-link diagrams. These node-link diagrams, or *graph drawings*, can help us see patterns and anomalies in graph structures [29].

To draw graphs, we turn to the specialised field of *graph drawing*, which goal is to create algorithms that produce drawings of graphs that are easy to follow [30]. The input of a *graph drawing algorithm* (also called *graph layout algorithm)* is a graph and the output is its *layout*. The layout contains information about where nodes are placed in the display space. There are two parameters, in addition to the input (a graph) to a graph drawing. The first, is its *convention*, rules that apply to the drawing. The second, is the *aesthetic criteria*, visual properties of a graph that a given graph drawing algorithm tries to achieve as best as possible. Both these parameters are covered later in this section.

Furthermore, there are many challenges in graph drawing. In the survey by Herman et. al. [31], the authors point out several issues that needs to be considered when visualising graphs:

- Size
- Predictability
- Time complexity

When graphs grow in size, the density of vertices and edges in the display space increases [31]. For dense graphs, this has an impact on the viewability and usability of a graph [31], assuming its part of a visualisation tool. Selecting individual nodes or links might be difficult, because of crossing edges and overlapping nodes [31]. Predictability is another factor. What this means is that applying the same layout algorithm on a graph, should not yield radically different visualised graph structures [31]. If a layout algorithm is unpredictable, this can break the cognitive map the user has learned when previously having viewed the same graph [32]. Last, there is the issue of time complexity for algorithms that produce graph layouts [31], presented later in this section. Some algorithms do heavy computations to place vertices, trying to achieve a set of aesthetic criteria. In this thesis, we use "big O notation" to express time complexity in graph drawing algorithms. The needed graph terminology, common aesthetic criteria, common graph drawing conventions, graph drawing approaches and a set of graph drawing algorithms relevant to this thesis are presented in the next few sections.

2.3.1 Graph terminology

Some basic graph terminology is needed to understand the graph drawing algorithms we describe later in this thesis. Next, we introduce some select terminology as presented in [33], Chapter 1 and Chapter 2.

A graph, $G = (V, E)$ is a pair of sets, a set of vertices, V and a set of edges, E, connected together. Vertices are sometimes referred to as nodes, and edges referred to as links. The *order*, or number of edges in a graph G is $|E|$ and the number of vertices is $|V|$. The vertices on each side of an edge, e is u and v. Both u and v are *adjacent* or *neighbours* to each other if connected by an edge, e. The edge e is *incident* to u and v. Two edges are adjacent to each other, if they share a common vertex. The *degree* of v is the number of its neighbours. If all the vertices of G are adjacent, the graph is *complete*.

A graph is *directed* if there exists a direction from an initial vertex $init(e)$ to a terminal vertex $ter(e)$ for every edge e in E. A graph with no direction,

is an *undirected* graph. Graphs may also have more than one edge between two vertices, these edges are referred to as *multiple edges*.

The *matching* problem is about finding a set of independent edges in a graph, where no edges share the same vertex. Vertices incident to the independent edges are considered *matched*. All other vertices are considered *unmatched*.

2.3.2 Conventions

The first parameter to a graph drawing, is the convention. Conventions are rules that graph drawings need to adhere to [30]. These rules may be that the drawing has to conform to a specific shape, or there are certain constraints set to how the edges are drawn. Battista et al. [30] list some widely used graph drawing conventions, where we introduce the most common here, namely:

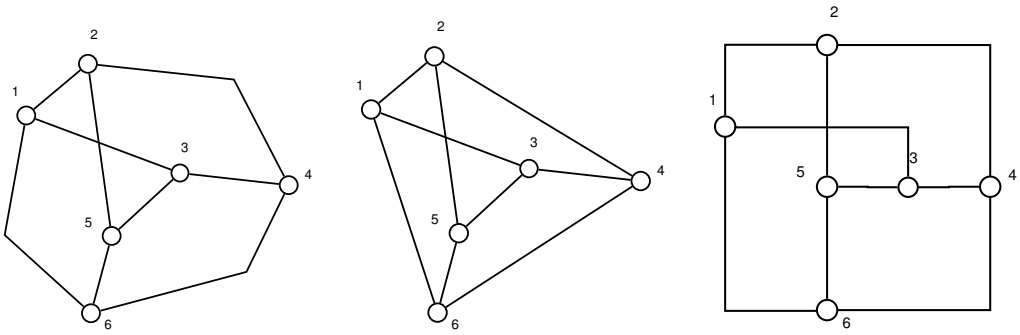
Polyline drawing Edges can have multiple bends. See Figure 2.12a.

Straight-line drawing Each edge is drawn as a straight line. See Figure 2.12b.

Orthogonal drawing Each edge is drawn as either horisontal or vertical edges. See Figure 2.12c.

Planar drawing No edges cross each other.

Polyline drawings are highly flexible, since one can create curved edges by increasing the amounts of bends [30]. However, polyline drawings with edges that have few bends (2-3) are hard to follow by eye [30]. Straight-line drawings are the most common and are featured in almost all literature mentioned in this thesis, mainly due to the visual complexity of polyline drawings. Orthogonal drawings are often used in circuit schematics and software engineering diagrams. Planar drawings are the most aesthetically appealing, being able to fulfill many of the aesthetics mentioned in Section 2.3.3. It is worth knowing that not all graphs are able to be drawn in such a way, such as the graph in Figure 2.12.



(a) Polyline convention (b) Straight-line convention (c) Orthogonal convention

Figure 2.12: Three different graph drawing conventions on the same graph. Adapted from Figure 2.1 in [30].

2.3.3 Aesthetics

The second parameter for graph drawing is the aesthetics. Graph drawing algorithms usually select a few aesthetics, that they try to apply as much as possible to improve readability of a graph [30]. The most common aesthetics are presented below [30]:

Edge crossings Try to minimise the amount of edge crossings.

Drawing area Try to minimise the drawing area to create area-efficient drawings, saving display space.

Edge length Try to keep edges the same length, alternatively set a lower and upper bound on edge length.

Edge bends Try to keep edge bends at a minimum, as these reduce readability of the graph.

Symmetry Try to keep graph as symmetrical as possible

Battista et. al. notes that "aesthetics often conflict with each other", so there are always tradeoffs [30]. As an example, having uniform edge length might conflict with trying to keep the drawing area as small as possible, if the edge lengths are uniform, but long. In addition, in the cases where

two aesthetics do not conflict, it might be algorithmically complex to handle both at the same time. Furthermore, the paper by Purchase et. al. [28] evaluated the aesthetics of edge crossings, edge bends and symmetry. The authors came to the conclusion that increasing edge bends and edge crossings in the drawing would decrease understandability of the graph [28]. Symmetry was deemed important, but there were no conclusive results that increasing symmetry would increase understandability of a graph [28]. When we later in this thesis decide on how to visualise network topologies, we have the three aforementioned aesthetics foremost in mind.

2.3.4 Approaches

Earlier, we established that the parameters that form a graph drawing, is one or more conventions and a set of aesthetics. From these, methodologies have evolved to conform to the different conventions and aesthetics of graph drawing [30]. The methodologies, or approaches as we call them, "divide the graph drawing process into a sequence of algorithmic steps, each one targeted to satisfy a certain subclass of aesthetics" [30], page 18. The issue of choosing an approach is also tied in to the application area of the graph one wants to draw. Factors like size, predictability and time complexity mentioned in Section 2.3, also come in to play. The most popular of these approaches, that are most suited to visualise the network topologies introduced in Section 2.1.1 are briefly mentioned and illustrated in general terms, below.

2.3.4.1 Orthogonal

Orthogonal approaches are often used in software engineering to show for example flow or entity-relationship diagrams [30], or by electronic engineers to draw circuit diagrams. This approach, typically uses a polyline and orthogonal convention, while trying to treat the different aesthetics equally [30]. It typically consists of a planarisation step, orthogonalisation step and a compaction step [30]. The planarisation step tries to reduce the number of edge crossings as much as possible, crossing are marked by a dummy vertex to create a planar drawing. The orthogonalisation step applies the orthogonal

convention to the graph. The compaction step tries to reduce the drawing area of the graph as much as possible. The process is outlined in Figure 2.13.

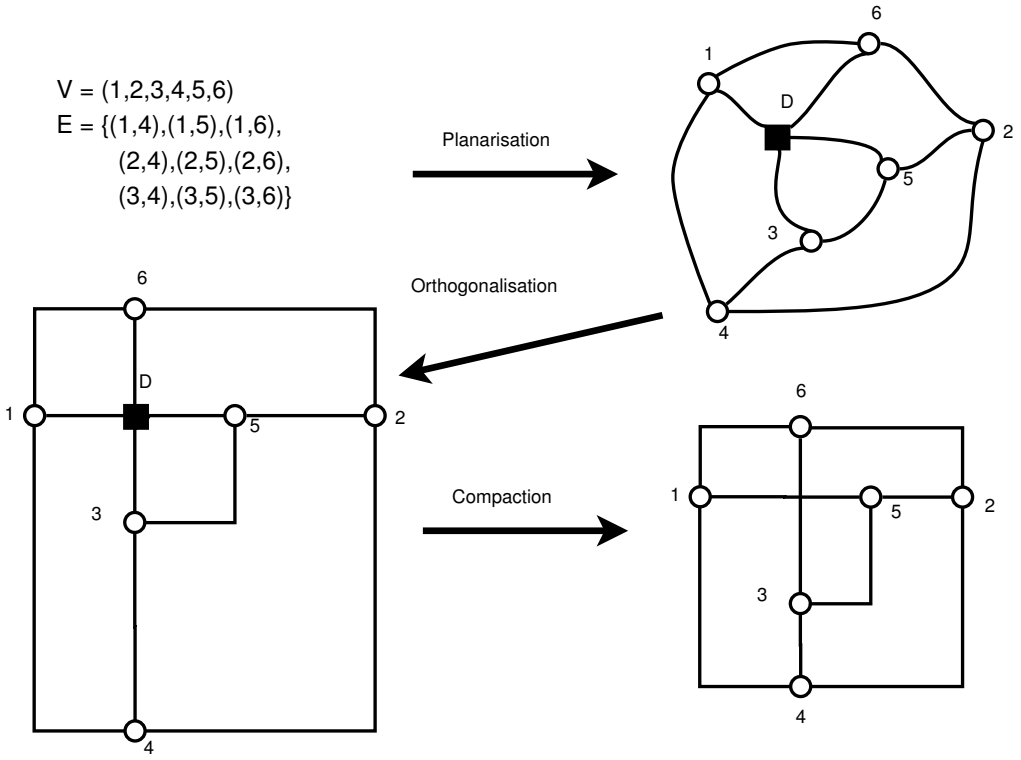


Figure 2.13: Orthogonal approach process. Adapted from Figure 2.6 in [30].

2.3.4.2 Hierarchical

As the name implies, hierarchical approaches are applied to graphs that imply a hierarchy, such as inheritance diagrams. It is typically used by directed graphs, to show dependencies, but can also be used on undirected graphs, by preprocessing the undirected graph so that its models a directed graph [30]. Hierarchical approaches often use a polyline convention. They are also quite intuitive to read [30]. Drawing a hierarchical graph involves three steps, layer assignment, crossing reduction and x-coordinate assignment [30]. The layer assignment step assign vertices to vertical layers. The crossing reduction step orders the vertices at each layer, trying to reduce crossings. The x-coordinate step, preserves the ordering of the vertices from the previous step and assigns

the final x-coordinates of the vertices, at each level. The process is outlined in Figure 2.14.

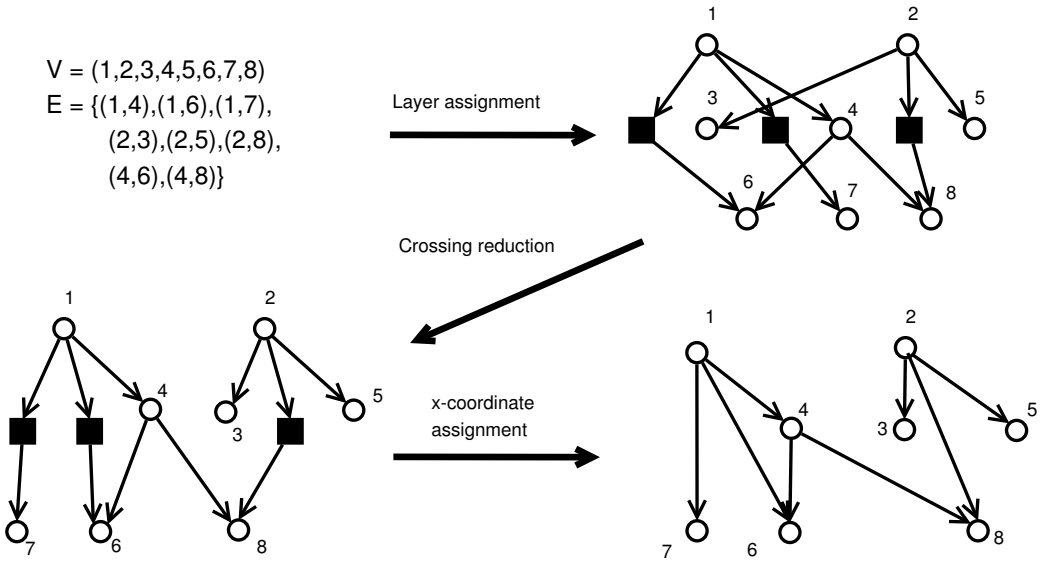


Figure 2.14: Hierarchical approach process. Adapted from Figure 2.7 in [30].

2.3.4.3 Force-directed

The force-directed approach, draws inspiration from the field of physics [30]. Algorithms using this approach, usually simulate "a system of bodies with forces acting between the bodies" [30], page 303. The goal is to generate a layout, where the vertices have converged to a state of local minimum energy configuration, also called *equilibrium configuration*. Force-directed approaches most often use straight-line drawing conventions, as these are easy to model [30].

There are two main components in this approach: A force model and a technique for finding the equilibrium configuration [30]. Examples of force models is a spring system, where vertices are modelled as balls and edges as springs. Another example is a spring-electrical system, where vertices are particles of equal polarity, repelling each other and edges modelled as springs, pulling particles together. Other force systems include the use of magnetic fields and annealing (heat treatment of materials).

The most common technique for finding the equilibrium configuration is by iterating- doing small changes to the layout, until it reaches a state where all vertices have zero energy [30]. See Figure 2.15 for an illustration of a spring system, with an iterating reduction technique. Something worth noting, is that the number of iterations needed to achieve the equilibrium configuration depends on the algorithm.

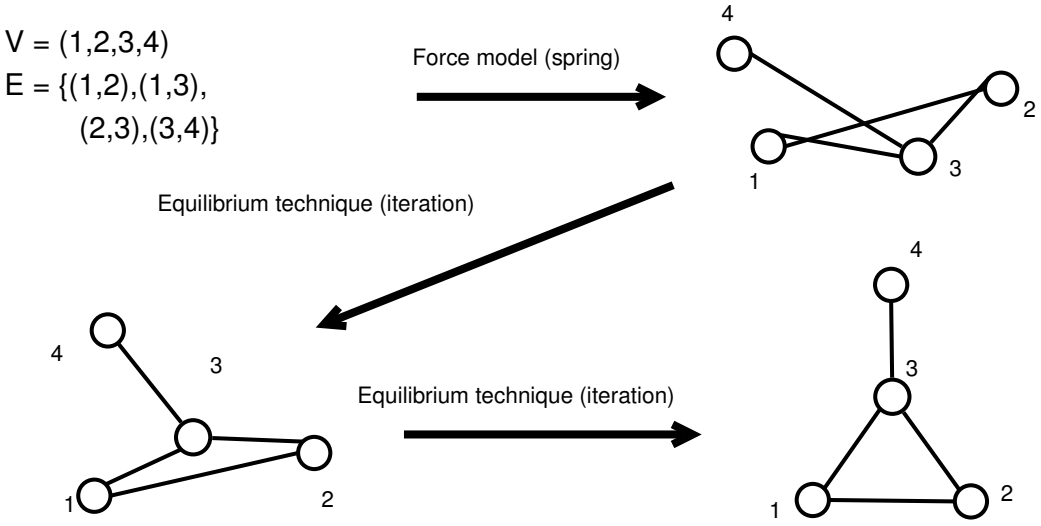


Figure 2.15: Force-directed approach using springs.

2.3.5 Force-directed algorithms

Describing all of the graph drawing algorithms available, for each approach, is outside the scope of this thesis. For an overview, one can refer to the surveys by Herman et. al. [31] and Battista et. al. [34], the book by Battista et. al. [30] and the website by Tamassia [35].

At first glance, it would make sense to use an algorithm with an orthogonal approach to draw meshes (Figure 2.3) and toruses (Figure 2.2), and a hierarchical algorithm to draw fat-trees (Figure 2.4). However, those are not the only topologies we encounter. In addition, we must also take into account irregular topologies.

A solution to this problem, would be to handle each topology different, applying a specific algorithms based on the topology, and apply a general one

to irregular topologies. This is quite tedious to implement and requires systems in the visualisation tool, to detect topology types, which in itself might be a complex and time consuming task. In addition, network topologies used in real-life applications are rarely 100% regular. Therefore, we have decided to focus on graph drawing algorithms using the force-directed approach, in this thesis. There are several reasons for this:

- The analogy to physics is easy to understand [30]
- They are easy to implement [30]
- They produce good layouts [30]
- They exhibit symmetries well [30]
- Many algorithms are made for 2D, but they can be generalised to 3D [31]
- They can be applied to general undirected graphs [31]

Drawbacks of the force-directed algorithms is their complexity [30,31] and unpredictability [31]. Certain implementations might have a complexity of up to $O(|V|^3)$ [36]. In addition, many algorithms do not give good layouts when the number of vertices increases. Battista et. al remarks that the algorithm of Frick, Ludwig and Mehldau [36] *"is one of the few methods that can handle graphs with more than 100 vertices"* [30], page 323. The book by Battista et. al. was written in 1998. After the year 2000, several techniques extended functionality of existing force-directed algorithms, to graphs with a thousand vertices or more [37]. The most common technique is the *multilevel* layout technique [37]. We explain this technique when we describe the multilevel force-directed placement algorithm, in Section 2.3.5.2.

There are many force-directed drawing algorithms that have been proposed. One of the first examples, is the SpringEmbedder by Eades [38], using spring mechanics similar to Hookes Law [31,37]. The algorithm by Eades has been revised and improved several times by authors such as Kamada and Kawai [39], Fruchterman and Reingold [40], Frick, Ludwig and Mehldau [36],

Davidson and Harel [41], Walshaw [29], and many more [30, 31]. Their main difference, is the physics model used, and what method that is used to achieve the equilibrium configuration [30].

In the next two sections, we show two force-directed placement algorithms. The first, the algorithm by Fruchterman and Reingold is based on the SpringEmbedder by Eades [40]. The second, the multilevel force-directed placement algorithm by Walshaw [29], uses a multilevel approach in positioning vertices, using a modified Fruchterman-Reingold algorithm to calculate the force displacement of vertices [29].

2.3.5.1 Fruchterman-Reingold algorithm

The algorithm of Fruchterman and Reingold uses a physical model of the movement of particles, where neighbouring vertices attrach each other and all other vertices repel each other [40]. The technique for finding the lowest energy state is by iteratively displacing vertices [40]. This is done, by using a "temperature" scheme, limiting the displacement of a vertice to a maximum value, decaying or "cooling" this value for each iteration [40]. The iterations continue, until the layout has converged to an equilibrium configuration and all vertices are in a state of zero energy. Each iteration has three steps:

- Calculate attractive forces on each vertex
- Calculate effect of repulsive forces
- Limit the total displacement by cooling the temperature

The initial configuration of the algorithm is done by randomly placing vertices. The algorithm is shown as pseudocode in Figure 1 of [40].

2.3.5.2 Multilevel force-directed placement algorithm

The multilevel force-directed placement algorithm of Walshaw, uses a multilevel framework in addition to a force-directed placement algorithm for laying out a graph [29]. A series of coarser graphs are constructed from the original graph by reducing the graph until the number of vertices are two [29]. The

process of *graph coarsening* is done by finding the maximal independent edge set of a graph (also known as maximal matching), where edges are collapsed to create a coarser version [29]. Walshaw uses the edge contraction method by Hendrickson and Leland [42]. Where, a randomly ordered list of vertices are visited in turn, matched to a random adjacent vertice, and collapsed to form a new vertex [42]. The neighbours of the new vertex is the union of neighbours of the collapsed vertices [42]. Each new vertex is also added a weight, which is the sum of the degree of the collapsed vertices [42].

After having done this, the two vertices of the coarsest graph is randomly laid out [29]. Each coarsened graph is then refined, placing clustered pairs of vertices at the same position as the cluster, until it has been extended to the original graph [29]. The force-directed algorithm applied to each refinement step is a modified version of the Fruchterman-Reingold algorithm, described in Section 2.3.5.1. The main modification in Walshaw's algorithm compared to that of Fruchterman-Reingold, is the repulsive force function, using a constant modifier C and the weight of the vertex as input [29]. The algorithm is shown as pseudocode in Figure 1 of [29]. This is the algorithm that we intend to implement in our visualisation tool prototype and the implementation details are described in Section 5.3.3.

2.4 Summary

In this chapter, we have presented an overview of interconnection networks, topologies, routing and the InfiniBand industry-standard. We have also presented key issues in information visualisation. We have given an introduction to the field of graph drawing, and presented some algorithms to achieve readable graph layouts. In the next chapter, we take a look at some visualisation tools for the presentation of network topology and simulation data in computer networks.

Chapter 3

Related work

This chapter presents an overview of tools related to the visualisation of network topologies and simulation data, describing their main features and visual look. Later, we see how common elements in these related visualisation tools have impacted the design and implementation of our prototype. Both in regards to visual communication and underlying implementation details.

The visualisation tools fall in several categories. Some are tightly integrated in the simulation environment, such as OMNeT++ [4, 5]. And some are external tools that parse topology and/or simulation event data such as NAM [7, 43], Interactive NS-2 protocol and environment confirmation tool (iNSpect) [44] and Huginn [45]. OMNeT++ has an option for visualisation integrated in its simulation environment. To give a broader perspective, we have described NS-2 [3] and three options for that simulator in Section 3.1. Both OMNeT++ and its visualisation is briefly introduced in Section 3.2.

Simulation data can also be shown in the form of plots or charts. Common tools for this is plotting programs (gnuplot, xplot), spreadsheet tools (OpenOffice Calc, Microsoft Word) or other data analysis frameworks. There also exist general network analysis tools (Cytoscape [9], Gephi [8], Pajek [10]) used for a broad range of research fields including computer networks, social networks, genome research and molecular research. Although we do draw inspiration from these, they are general graph analysis tools, unable to visualise

simulation data.

3.1 Network Simulator 2

NS-2 [3] is a discrete-event simulator aimed at network research, based on the REAL network simulator. It supports simulation of transport-, routing- and multicast- protocols over both wired and wireless networks. NS-2 is implemented using two languages: the object-orientedc simulator environment written in C++ and the OTcl (an object-oriented extension of Tcl) interpreter that acts as a front-end. Both implementations have their own associated class hierarchies, the C++ class hierarchy (compiled hierarchy) and the OTcl class hierarchy (interpreted hierarchy), that are mapped one-to-one. NS-2 can be extended to simulate other protocols or event-driven scenarios not covered by the class libraries [46].

Listing 3.1: Sample NS-2 trace file [46], showing packet transmission events.

```
+ 1.84375 0 2 cbr 210 ------- 0 0.0 3.1 225 610
- 1.84375 0 2 cbr 210 ------- 0 0.0 3.1 225 610
r 1.84471 2 1 cbr 210 ------- 1 3.0 1.0 195 600
r 1.84566 2 0 ack 40 ------- 2 3.2 0.1 82 602
+ 1.84566 0 2 tcp 1000 ------- 2 0.1 3.2 102 611
- 1.84566 0 2 tcp 1000 ------- 2 0.1 3.2 102 611
r 1.84609 0 2 cbr 210 ------- 0 0.0 3.1 225 610
+ 1.84609 2 3 cbr 210 ------- 0 0.0 3.1 225 610
d 1.84609 2 3 cbr 210 ------- 0 0.0 3.1 225 610
- 1.8461 2 3 cbr 210 ------- 0 0.0 3.1 192 511
r 1.84612 3 2 cbr 210 ------- 1 3.0 1.0 196 603
+ 1.84612 2 1 cbr 210 ------- 1 3.0 1.0 196 603
- 1.84612 2 1 cbr 210 ------- 1 3.0 1.0 196 603
```

NS-2 outputs event data, that can either be displayed on-screen while the simulation is running or dumped to files called trace files. Contained within the trace files, information about topology, layout and events can be disseminated (one line in the trace file is an event) Trace files form the basis for analysis and can be used as input to a visualisation tool. Listing 3.1 shows a section of a sample NS-2 trace file, where each line is an event

identified by the first character of the line. “s” being a packet send event, “r” being a packet receive event and “d” being a packet drop event. In the following sections we give a brief description of what visualisation options that exists for NS-2.

3.1.1 Network Animator

NAM [3, 7] is an animation tool for network simulation traces and packet data, based on the Tcl/Tk widget toolkit and is the default visualisation tool for NS-2 [46]. It can take NAM trace files as input, that contains information about network topology and packet events. Alternatively, the user can interact with NAM to create and edit network topologies to use with NS-2. In addition, NS-2 can generate a NAM trace file from a simulation run. NAM is not entirely dependent on NS-2 to function, other programs can also generate NAM trace files for animation in NAM.

NAM has five basic building blocks when visualising simulation events. As seen in Figure 3.1, the building blocks are [7]:

Node Represents a switch, router or host. Can have three shapes that are immutable: circle, square and hexagon. Can be labelled and change colour during animation.

Link Represents a link between nodes. A full-duplex link, modelled as two bidirectional simplex links. Can be labelled and change colour during animation.

Packet Represents a packet in a network. Shown as a block with an arrow, if in a queue it is shown as a square. If the packet is dropped, it is shown as a falling rotating square (not visible during backward animation). Can be coloured.

Queue Represents a packet queue. Connected to a simplex link in a link animation object. Shown as packets stacked on a line, which can be orientated in an angle from the horizontal line.

Agent Represents protocol states in nodes. Shown as a square with name inside, next to a node.

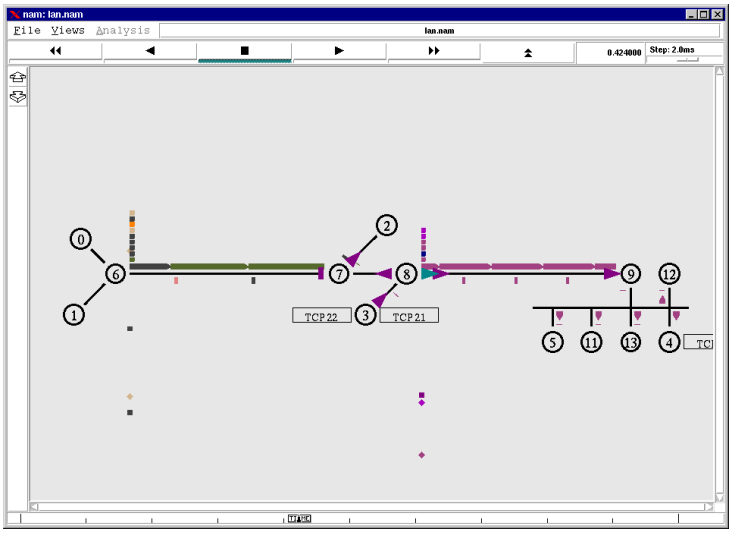


Figure 3.1: An image showing the building blocks of NAM [43].

NAM offers a 2D cartesian (x, y) layout and three ways of doing layout of network topologies. Manual placement (x,y) of nodes, manually setting the link-orientation between two nodes and automatic layout. The automatic layout method works by modelling nodes as balls and links as springs. Connected balls repulse each other and springs pull them together. After a number of iterations, the layouts ideally converge. This layout method is similar to a force-directed approach, introduced in Section 2.3.4.3. The manual for NAM suggests automatic layout with hand placement for large graphs and link orientation for smaller ones [46].

The user can interact with NAM by zooming in and out, manipulating a timeline for rewinding, playing backwards and playing forwards in various speeds. It has a time label for showing where in the simulation timeline the animation is at. The user can drag a slider to skip to different times in the timeline. In addition, NAM has a box where events are described as (time, string) pairs, called annotations. The user may have annotated the trace file using NS-2 or may add, edit or remove them manually during a simulation, whereas they are recorded in the trace file. The user can also observe packet

information in a monitor window, get a popup to see bandwidth utilisation or show a loss graph when selecting simplex edges [46].

To store the playback of a NAM animation, users can save individual files on a per-frame basis, which can later be post-processed into making animated .gifs or MPEG vids using other tools. NAM can also read from filestreams such as STDIN to give a near-realtime animation of a simulation, piped from a ns-2 simulation execution [46].

Due to the almost nonexistent wireless support, other tools have been developed as visualisation support for NS-2 [44, 45]. These tools fill the gap in NAM functionality. Another drawback is the flexibility of NAM, since the animation complexity is pushed on the user. In the next section we look at a tool that supports wireless networks.

3.1.2 Interactive NS-2 protocol and environment confirmation tool

iNSpect [44, 47] is a cross-platform Open Graphics Library (OpenGL) [48] based visualisation tool written in C++ for NS-2 wireless simulations. It was made in a time when NS-2 had recently been extended with wireless network support, but NAM was not. The difference from NAM is that iNSpect can read mobility files (input to NS-2) and post-simulation NS-2 trace files with no modifications. Optionally one can use the custom iNSpect vizTrace format [49].

iNSpect draws nodes on a 3D plane (originally it was 2D [44]). As shown in Figure 3.2, nodes are shown as balls that can change colour depending on the state. The colour of a node is important for event feedback [50]. After a transmission is initiated, the source node is coloured blue, the receiving node coloured red and the final destination coloured green. Other colour codes can be customised. For example: a node receiving a duplicate packet might be coloured orange. During transmission, a black line is extended from the source node to the destination node. Overlay objects like circles or rectangles can be added transparently on top of the nodes. These can denote for example congested areas (work cantina), physical obstacles (concrete building) or

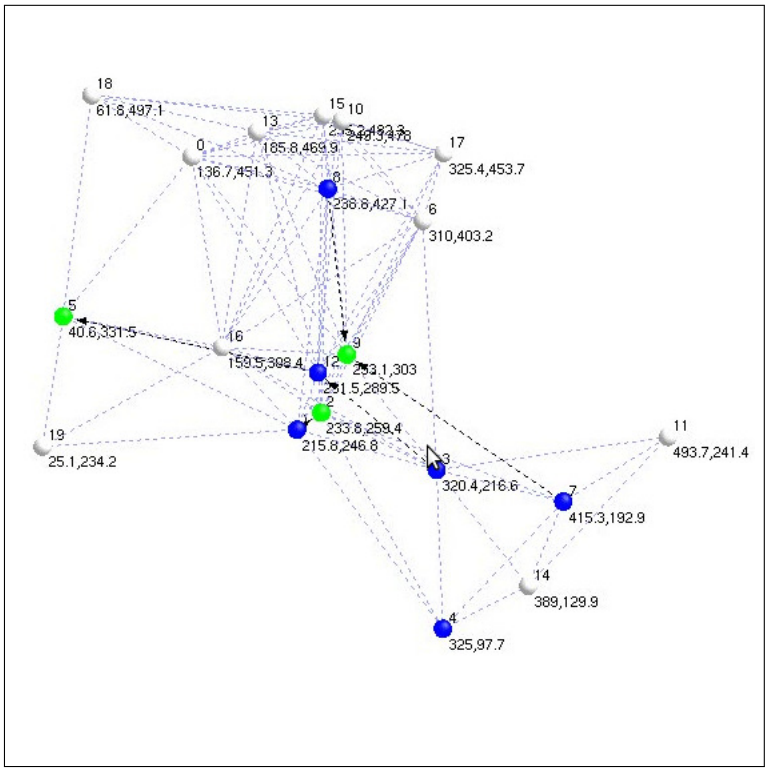


Figure 3.2: An image of iNSpect showing node positions, graph connectivity and node states [47].

other areas of interest. A background image can be added to give context to the simulation, for example a sketch of an university seen from above where mobile devices are shown as nodes in different rooms [49].

The user can navigate the scene by zooming and panning. There is also a slider for simulation playback with buttons to pause, resume, play forwards and backwards at various speeds. A drawback is that there does not seem to be any option to jump to an arbitrary position in the timeline. Selecting a node yields a transmission ring, a dotted circle around the node denoting ideal transmission range. Clicking a node also updates a node status window, showing location, status, id and node report of the selected node. Node reports shows statistical data such as number of packets received, dropped and forwarded. The user can take screenshots in png or ppm formats or capture animation in MJPEG format, with controls for setting start time, stop time and capturing frame rate [49].

Features for data analysis have been added in newer versions of iNSpect. A graph connectivity tool renders lines between nodes based on the transmission range of nodes, this can be used to verify correctness of node behaviour and visually show shortest path, unavailable paths and routing loops. A graph partitioning tool has also been added, it is used for changing the appearance of nodes that move out of range and can impact protocol performance. An overlay for showing node locations can also be activated by clicking a button, it can be used to verify if the simulation program did the correct movements or placements compared to what the simulation input dictated [49].

One can supply a configuration file for iNSpect, prior to execution. This minimises command-line arguments and gives flexibility to control various display elements, such as position and size of overlay objects and colour codes. The supplied configuration file contains default values that can be customised. In other words, the user can define visualisation in iNSpect to highlight the most useful data.

3.1.3 Huginn

Huginn [45,51] is a toolbox for visualising Mobile Ad Hoc Network (MANET) traces from NS-2. It was created as a response to the lack of wireless support in NAM. Scheuermann et al. states that visualisation tools can greatly aid researchers in identifying and avoiding mistakes in protocol development [45]. Huginn is not a single software component in the classical sense i.e. linked into a single binary. But an UNIX pipeline consisting of three main parts: Flowchart Editor (FE), Evaluation Engine (EE) and Visualisation Engine (VE). Each of the components communicate with each other by passing messages or sharing configuration scripts.

The FE is implemented in C++ and is used for all interaction tasks prior to visualisation. It gives the user the option to filter, aggregate, scale and display each event or state that might be interesting. For example, a user interested in the number of packets received per node, can aggregate packet count, scale the value logarithmically and display a barchart above the node

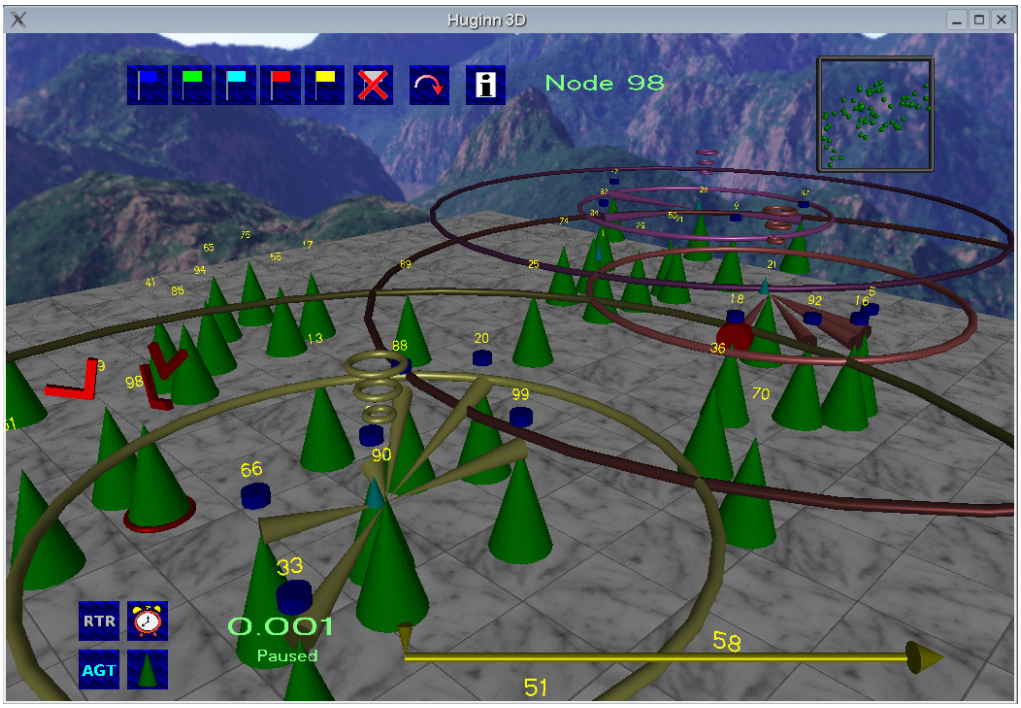


Figure 3.3: An image of Huginn showing nodes, transmission and carrier sensing ranges [51].

during visualisation. The user needs to have knowledge of NS-2 wireless trace file semantics before working on the flow chart. Flow chart files are re-usable for any trace file sharing the same configurations [45].

The EE, written in Ruby does not interact with the user. It reads the configuration flowchart and applies the evaluations to the given NS-2 wireless trace file, which is input to the VE. The EE also holds necessary data structures for the VE during run-time. The EE and VE communicate using two-way Inter-process communication (IPC) and a minimum of high-level communication is needed. This is because the VE knows how the scene looks like and the EE knows the data structures that composes the scene, only changes need to be communicated. The EE initiates the VE, which we have describe above [45].

Huginn has a VE as the last component of the pipeline. It uses a high-level configurable visualisation engine library called V-IDS to draw a 3D scene. As seen in Figure 3.3, the scene is drawn as a two-dimensional MANET

scenario. The third dimension is used to show the inner state of nodes, packet transmission process and statistics. Nodes are shown as cones and each cone is in the centre of two circles, where one circle shows the transmission range and the other circle shows carrier sensing range. This is so the user can see if two nodes are blocking each other. Transmission of packets are visualised as horizontal cones stretching from one node to the other. The base of this horizontal cone acts as the sending node and apex the receiving node [45].

Navigating the scene can be done with the mouse or a joystick, with features like zooming in and panning the scene for a closer look at areas of interest. A time line at the bottom of the scene, shows the position in simulated time. It has two modes: linear time and FlexTime. Linear time allows the playback of the simulation at various speeds. FlexTime adapts visualisation speed to the amount of events happening in certain time intervals. For example, if there is a period of inactivity, the visualisation speeds up. One can click on the timeline to jump to the corresponding time point in simulation time [45].

3.1.4 Summary

In this section, we have briefly mentioned NS-2 and described three visualisation options made for it. We have seen that NAM fulfils many of the basic needs required by wired network simulations. Two of the tools we have seen were originally developed because of the lack of wireless support in NAM. In the next section, we look at OMNeT++ and its options for visualisation.

3.2 OMNeT++ and visualisation

OMNeT++ [4, 5] is an object-oriented modular discrete-event simulation framework. It is a simulation package with the ability to simulate just about any scenario involving discrete events, but is targetted at computer networks and distributed systems [4]. OMNeT++ has a component architecture for simulation models, where models are formed from modules. Modules are written in C++ and can be connected and combined to form compound

modules. The OMNeT++ manual compares this process of "moduling" to building with LEGO blocks [52]. Examples of modules is a node consisting of several nodes. The Network Description (NED) language of OMNeT++ is where the set-up of network simulation takes place and forming of compunded modules occur. OMNeT++ ships with the Eclipse Integrated Development Environment (IDE) for editing NED and module files both programatically and graphically [52]. OMNeT++ has the option to output its own flavour of simulation trace files, called event logs. A sample event log is shown in Listing 3.2, where each line is an event identified by the code at the start of the line. For example, "BS" code indicating a message send event and "CE" being a message cancel event. OMNeT++ also supplies tools for the statistical analysis of event logs. For more information about the OMNeT++ event log grammar and semantics, we refer to the OMNeT++ manual, Chapter 25 [52].

Listing 3.2: Sample of an OMNeT++ event log [52].

```
E # 14 t 1.018454036455 m 8 ce 9 msg 6
BS id 6 tid 6 c cMessage n send/endTx pe 14
ES t 4.840247053855
MS id 8 d t=TRANSMIT,,#808000;i=device/pc_s
MS id 8 d t=,,#808000;i=device/pc_s

E # 15 t 1.025727827674 m 2 ce 13 msg 25
- another frame arrived while receiving -- collision!
CE id 0 pe 12
BS id 0 tid 0 c cMessage n end-reception pe 15
ES t 1.12489449434
BU id 2 txt "Collision! (3 frames)"
DM id 25 pe 15
```

OMNeT++ has a built-in visualisation option that is executable via the IDE or via the command-line [52]. It does not use any event log file from OMNeT++ to parse and run post-simulation. It is rather a part of the simulation process, where the user can step-by-step see the simulation execution in real-time. It is done by a Tcl/Tk based user-interface called Tkenv. When the simulation is initiated, the simulation controls showing the details

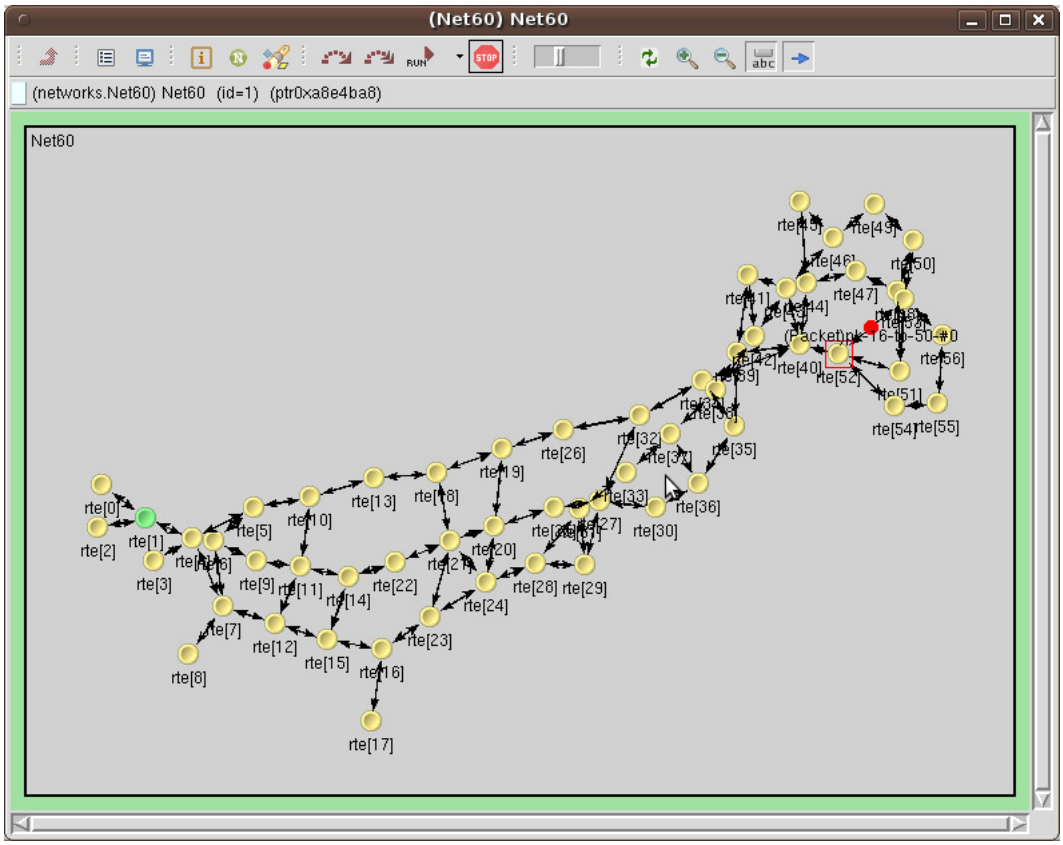


Figure 3.4: An image of an OMNeT++ Tkenv sample.

of simulation execution opens up in a window. In addition another interface window is opened, displaying the visualised topology. The display scene is in 2D (x, y) where the topology is assembled and viewed. The NED file contains information about what the topology should look like with a @display property [52].

The visual look in Tkenv is completely user controlled pre-simulation [52]. Rather than having hard-coded visual figures for nodes like in NAM, iNSpect and Huginn. One can import just about any graphical icon or geometrical shape of various colours in Tkenv. Connections between nodes are shown as lines in wired scenarios or as a transmission ring for wireless scenarios. Background images can be added to give context to the visual simulation, just like in iNSpect. Due to the high degree of customisation it is recommended to view the OMNeT++ manual, chapter 10 for further information [52]. Please

refer to Figure 3.4 to see an example of OMNeT++ in action. Just like NAM, Tkenv pushes alot of the animation complexity on the user.

Nodes in Tkenv can be manually laid out using a tag in the @display property of a module. If this is omitted, Tkenv attempts to automatically layout the network topology using the SpringEmbedder algorithm of Eades [38]. The manual notes that it can produce "funny" or "erratic" results depending on the number of nodes or size of modules [52]. The user can re-layout the topology by clicking a button. This layout scheme is similar to the one used by NAM and highly popular in many general network analysis tools [8–10].

Tkenv gives the option to zoom in and out to view certain areas in more detail and also inspect model information, component types, messages, queues, nodes, connections, etc. Since the simulation environment is running in parallel with the visualisation support, the latter has access to all the data structures needed to show almost every aspect of the simulation. For this reason, simulations do not seem to be able to be played backwards, nor does one have a timeline to choose specific timestamps that one might be interested in.

In this section, we have briefly described the default option for visualising simulation runs in OMNeT++. Recently, a visualisation extension for OMNeT++, called OmVis [53] has emerged. It offers a parallel multi-view approach to observe simulation runs both in space and time [53]. However, only one paper has been written about it and there is no binary, source code or documentation available as of yet. But it does show, that there is still a need for visualisation tools to aid in the development and analysis of protocols and algorithms prior to and after simulations.

The next section summarises this chapter.

3.3 Summary

In this chapter, we have given an overview of the visualisation options for NS-2 and OMNeT++. In the next chapter, we try to use what we have learned from these previous efforts.

Chapter 4

Design

In this chapter, we present the design of our visualisation tool prototype for viewing network topologies and OMNeT++ simulation data.

We begin this chapter by giving an introduction to the ideas behind the visualisation tool in Section 4.1. Next, we present the design overview for the visualisation tool in Section 4.2. We then move on to discuss the design of each main component in our prototype visualisation tool, in turn. Starting with the data model component in Section 4.3 and then the visualisation component in Section 4.4. Lastly, we summarise and discuss the design in Section 4.5.

4.1 Introduction

In Section 2.2, we saw that visualisations can help in gaining insight and understanding about problems. However, designing a visualisation tool is not a trivial task, as the user has to be informed by the visualisation. As Card et. al. stated: "The purpose of visualization is insight, not pictures" [6], page 6. Furthermore, as outlined in Chapter 3, there have been several visualisation tools created for viewing simulation data. Such tools gives the user a multitude of dynamic and interactive options to solve many tasks.

To better aid the researcher to post-analyse OMNeT++ simulation traces and OpenSM-defined network topologies, we propose a stand-alone prototype

visualisation tool, named *IBSimVis*. It is designed to offer a dynamic visualisation of network topologies and OMNeT++ simulation data. From this chapter, and the rest of this thesis, we refer to the prototype visualisation tool we have designed and implemented, as *IBSimVis*.

4.1.1 Usage scenario

As illustrated in Figure 4.1, the typical usage scenario is the following. The user wants to study a network topology or results from a simulation run. The user then inputs the network topology, routing table and simulation data to IBSimVis. IBSimVis then initialises and offers an interactive view of the topology, its entities and attributes. The user can then interact with the topology, changing views as needed, for both exploration and analysis.

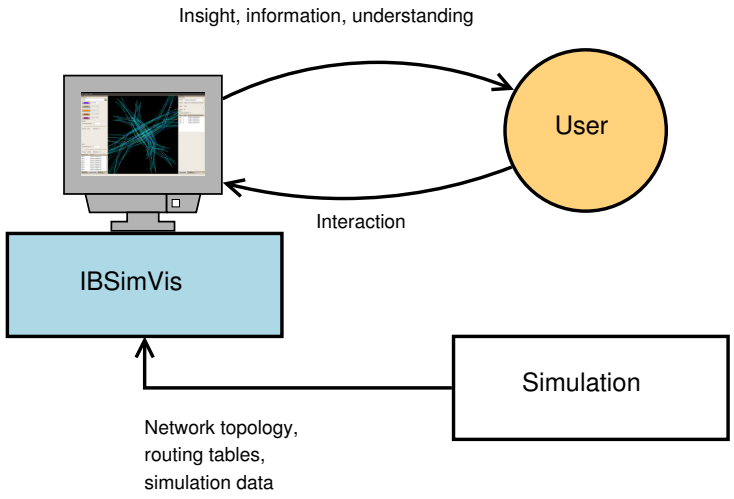


Figure 4.1: Usage scenario for IBSimVis, showing a simple usage flow, where data is input to IBSimVis, returning a visualisation for the user to interact with and gain information from.

4.1.2 Characteristics and requirements

There are many challenges in creating visualisation tools. But the information a user gains from using a visualisation tool, can be of great value. The

basis for our design requirements is the common characteristics that the related visualisation tools, introduced in Chapter 3 exhibit. Based on those tools, we have isolated the following characteristics:

1. Ability to view network topology in 2D or 3D
2. A strategy to layout the network topology
3. Ability to inspect entities and their attributes
4. Ability to differentiate between nodes, using various encoding mechanisms
5. Ability to view the packet transmission process
6. Ability to playback simulation data

Another characteristic, that had an impact on the design of IBSimVis, is that all related work, with the exception of visualisation in OMNeT++, are stand-alone applications. Stand-alone application means that the tool is not part of a framework or integrated in the simulation environment. The reason why we have created IBSimVis as a stand-alone tool is discussed in Section 4.2.

Due to the large amount of different simulation events occuring in simulation data, we have decided to limit the design of IBSimVis to only being able to visualise simulation message transmissions. That is, we have considered strategies to parse and detect events related to sending, forwarding, dropping and receiving messages. The same limitation exist in related work for the same reason. For example Huginn, iNSpect and NAM can only visualise transmission processes and inner state of network nodes. We leave it as future work to visualise all aspects of an OMNeT++ simulation event log.

From the aforementioned characteristics and by looking at the problem domain, we have extracted a list of functional requirements and features that we want IBSimVis to have. These requirements are, namely:

- Read topology data files, routing table files and simulation trace files

- Layout the network topology
- Provide presentation and interaction of network topology in 2D or 3D
- Provide the user with a GUI to filter data, manipulate simulation playback and change views

The requirements are not only consistent with the reviewed related work, but also general graph visualisation tools, that have only been mentioned briefly. Such as Gephi [8] and Cytoscape [9]. Bastian et. al. state that the most important requirements for a network exploration tool are: "high quality layout algorithms, data filtering, clustering, statistics and annotation." [8], coupled with a modern user-interface [8].

A key consideration when designing IBSimVis, was to give the user the ability to quickly get a visualisation of the network topology. So the user can immediately start to analyse and explore it. We want novices that are not acquainted to the topology or simulation scenario, and experts to easily get started with IBSimVis. To achieve this, we want an extensive GUI to go together with the topology visualisation for changing views and data filtering. Contrary to for example Huginn, requiring the user to pre-configure aspects of the visualisation using a flowchart editor. We also want the inner workings of IBSimVis, to be as transparent to the user as possible. This consideration is reflected in our design and implementation of IBSimVis.

Among the requirements, one of the major challenges is to be able to visualise the network topology. In related work, we saw that the central visual element of all the tools, was the ability to present a network topology. As mentioned in Section 2.1.1, not only do we want to be able to visualise both regular and irregular topologies, we also want IBSimVis to be able to scale and handle large networks. The size of a topology and the time complexity in layout algorithms pose serious design and implementation challenges. These challenges are discussed in Section 4.3.3. To alleviate the network size problem, when visualising a network topology, we want to explore the use of 3D, since it gives the ability to encode data in another dimension. The use of 3D also gives additional challenges to presentation and interaction, which we discuss in Section 4.4.1 and Section 4.4.6.

Furthermore, we also want to consider the users perception of a visualised topology. Consider Figure 2.1c. In literature, this is the standard approach in drawing Fat-trees, see [13] (Figure 1.23b) and [14] (Figure 2). This figure shows a tree with k root nodes, connected to k switch nodes in n levels, with k processing nodes, connected to the switches at the lowest level. What happens to the understanding a user has of a given Fat-tree, when the topological layout changes radically? It may destroy the "mental map" a user has of a topology [32], forcing the user to re-learn the topology. We discuss this in Section 4.3.3.2.

Section 4.3.1.2 addresses a parsing challenge, how IBSimVis handles the simulation, as it changes over time. This has an impact on our parsing strategy and communication between components, as the simulation model requires that enough events have been recorded in advance [45]. Another challenge is to correlate events, assuming enough data has been read in advance from the simulation trace file, discussed in Section 4.3.1.3.

In information visualisation, a crucial aspect is giving the user the ability to view data not only from different angles, but also in different contexts. As mentioned in the requirements, we refer to this as changing *views*. A view in IBSimVis is an user-selectable feature for viewing data in different contexts. Entities remain the same, but different encoding mechanisms may be applied to them, depending on the chosen view. For example, the normal view offers the standard glyphs, used for printing on paper or presentations. While, selecting the NumPath view, shows path distribution over the topology, giving the user the ability to see potential choke-points in the topology. Views are discussed in Section 4.4.7.

The characteristics and requirements introduced here, in addition to the select challenges we have mentioned above, all have an impact on our design and are discussed in this chapter. In the next sections, we introduce the design overview of IBSimVis and discuss design decisions of main components. We start by showing the design overview of IBSimVis.

4.2 Design overview

Early in the design process, we decided that IBSimVis should be a stand-alone application, seperate from the simulation environment. Alternatively, it could have been built on top of OMNeT++, offering real-time visualisation of simulations. We decided against this, because getting acquainted to the inner workings of OMNeT++ would have taken a long time and because we did not want IBSimVis to be tied to a specific IBA simulation tool, instead giving the option to work with other IBA simulation tools in the future.

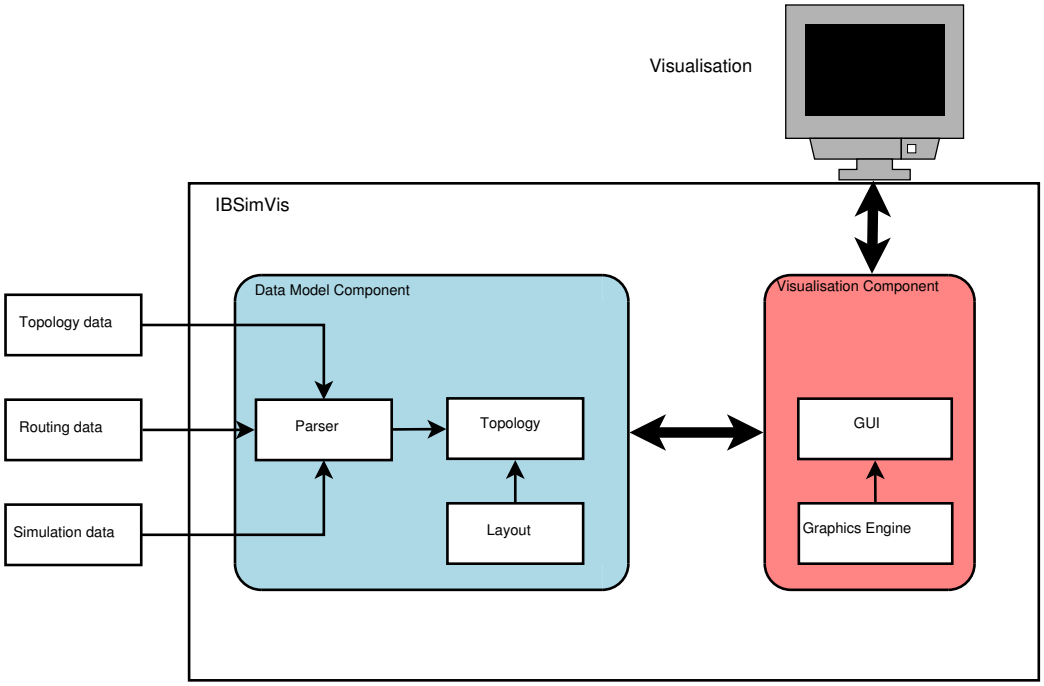


Figure 4.2: Design overview, showing components and inter-component communication.

IBSimVis is considered the *main application*, which controls all the internal components. The main application consists of two main components, as illustrated in Figure 4.2, namely: the Data Model Component (DMC) and the Visualisation Component (VC). Each main component addresses one or more of the issues in information visualisation: representation, presentation and interaction. The DMC contains all the entities, their representations and strategies for parsing data. In addition, it helps facilitate presentation

of a network topology. The VC takes care of presentation and interaction. Furthermore, each main component consist of subcomponents, each handling specific tasks, linked to the requirements mentioned in Section 4.1.2. For the DMC, the subcomponents are:

Parser Parses topology data, routing data and simulation data. A component such as the Parser is needed to create the entities that we want to represent and read simulation data.

Topology Main datastructure for IBSimVis. Keeps track of both static and dynamic entities. Calculates and maintains statistical data derived from both static and dynamic data, such as path distribution and packet counts. This statistical data is the basis for IBSimVis's ability to change views.

Layout Handles the presentation of graph structures, by calculating the layout of nodes, using force-directed drawing algorithms, covered in Section 2.3.5.

For the VC, the subcomponents are:

Graphics Engine Contains encoding mechanisms for entities and values in Topology. Able to render and display a visualisation of static topology in 2D or 3D.

GUI Handles interaction between the user and the visualisation. Gives the user the option to change to different views and filter values, both spatial and temporal in nature.

The reason for the division into only two main components is that as mentioned above, we wanted to keep the design as simple and lightweight as possible. Furthermore, each subcomponent handles specific tasks or issues connected to the requirements mentioned in Section 4.1.2. We also believe the component division is intuitive in regards to the information visualisation process, mentioned in Section 2.2 and shown in Figure 2.7.

Before initialisation, the user adds topology data, routing data and simulation data as parameters to IBSimVis. Topology data contains information about nodes, links and ports in the network topology, i.e. the data type is structural relations. Routing data contains the LFT for all switches in the subnet. Simulation data contains information about a simulation run. Furthermore, we refer to the topology data and routing data as *static data*. Simulation data is referred to as *dynamic data*. Depending on the simulator network model, simulation data can be different in every simulation run, even with the same set of static data. (e.g. links or nodes randomly dropping packets)

After intitialisation, the Parser identifies and creates entities based on the static data, by adding them to the Topology. After topology data has been read, it invokes the Layout component in an own thread, while continuing to read routing data in the main thread. The Layout component is invoked with the default implemented layout-algorithm. While routing data and layout is being applied on the static topology, the Parser commences to buffer simulation data, so that it is ready for display when the visualisation is ready. After layout has been applied to the static topology, the VC is invoked, by taking the Topology as a parameter and communicating to both its subcomponents to create the visualisation. An event loop is created, listening for user interaction and handling input appropriately.

Since IBSimVis is a stand-alone application, with only two main components. Communication between subcomponents is done via direct function calls, by passing object pointers or references. This makes communication overhead very low. Contrary to for example Huginn, which essentially consists of three different programs, with different human-readable text-based communication protocols between each step in the UNIX-like pipeline [54]. The advantage with this is that debugging is made easier [54] at the cost of communication overhead, such as parsing the text-based protocol. Thus, since we want IBSimVis to be as fast and lighweight as possible, we do not use the approach mentioned above and do not discuss communication between subcomponents in more detail. However, communication between the two main components, the DMC and the VC poses some design decisions,

covered in Section 4.4.5. Next, we move on to discuss the design choices and challenges in the DMC.

4.3 Data Model Component

In this section, we discuss the design choices made for the Data Model Component (DMC). We discuss issues and challenges we had to consider in each subcomponent, mentioned above. Next, we discuss each subcomponent in the DMC, starting with the Parser subcomponent.

4.3.1 Parser

The Parser component parses data, identifies and creates entities. The input to the Parser component are files, containing topology data and routing data from OpenSM. In turn, the data from OpenSM was created from simulating an InfiniBand fabric, using IBMgtSim. The input to the Parser component also include simulation data from OMNeT++. The format and semantics of these files are described later, when we describe how we implemented the Parser, in Section 5.3.1.

The Parser component has three main tasks:

1. Parse the topology data from OpenSM/IBMgtSim.
2. Parse the routing data from OpenSM/IBMgtSim.
3. Parse the simulation data from OMNeT++.

To reduce inter-component communication overhead, we want the Parser to directly communicate with the Topology for creating and manipulating entities. Thus when intiating the Parser, it takes the Topology as a parameter. This way, the Parser can call on the exposed interface of Topology to create and manipulate the entities directly with minimal overhead.

Parsing (or syntactic analysis [55]) of static data is a straight-forward task. Nodes, ports, links and their structural relationships are clearly defined in the topology file. In addition, LFT entries in the routing table file have a clear

relation to each switch. Parsing static data is only done once, at the initialisation step of IBSimVis. Parsing the OMNeT++ event log is made easier since the grammar of the event log is readily available in the manual [52] (Chapter 25), in Backus-Naur Form (BNF) [56]. Since the formal grammar for the event log is defined, we want to use a scanner to tokenise data input and send it to a parser generator to parse the dynamic data. Note that it is outside the scope of this thesis to cover parsing techniques. For a formal introduction to parsing and parsing techniques, we refer to the book by Grune et. al. [55]. Parsing simulation data still rise some issues. First, because the size of this data can be substantial and can not be kept in memory at the same time [45], even when abstracted into data structures with low memory footprint. Second, because the start-up delay of the visualisation can be long [45]. Third, because the semantic and structure of the simulation data file is not straight forward [45]. (e.g. there might be several thousand different, small events between a source node transmitting a packet and the sender receiving the packet) Among the reviewed visualiation tools introduced in Chapter 4, only Huginn and the article by Scheuermann [45] describes challenges in parsing simulation data. Thus, we often refer to strategies introduced in that article. In the next sections, we discuss issues and challenges around a general file format for IBSimVis and parsing the simulation data file.

4.3.1.1 General file format

As mentioned above, IBSimVis has to be able to handle three different file formats to visualise static and dynamic data. This is unavoidable when processing data for a given simulation run for the first time, as we have no control over the applications that IBSimVis depends on.

However, if the same topology and simulation is considered several times, a general file format for visualising the simulation run would be advantageous. A possible approach is used in NAM and iNSpect. As mentioned in Section 3.1.2, iNSpect has its own optional file format for showing visualisations, named *vizTrace* [49]. The main motivation for the vizTrace format was that MANET research used many different simulation tools [49]. The

vizTrace file format consists of six fields that describe each event to create the visualisation, in the form of:

```
[NodeID] [timestamp] [sending to|received from] [NodeID]
  [source|forwarding|destination] [PacketID]
```

The vizTrace file needs to be coupled with a file describing the topology, such as a mobility file [49], as such data is not present in the vizTrace format.

The advantages of this approach is that it minimises the overhead required to initialise and start the visualisation, as all events have already been disseminated and grouped. In fact, any simulation tool could output such a file format and IBSimVis would be able to visualise the simulation. The general file format would be smaller in size, than the combined size of the three input files to IBSimVis. The drawback is that a general file format in a form similar to the vizTrace format, would be too simple to be used as a basis for analysis and evaluation. For this reason and the fact that the amount of simulation tools used for InfiniBand networks are very limited. In addition, to time constraints, we opted against defining a general file format for IBSimVis, leaving it as future work.

4.3.1.2 Simulation data look-ahead

All static data is required to be read, for IBSimVis to understand what the network topology looks like. For simulation data, however, we do not want to pre-process the simulation data prior to visualisation. As mentioned earlier, because the start-up delay is increased by parsing and analysing possibly millions of simulation events [45]. And second, because of excessive memory consumption [45]. Therefore, we need an approach to both parse and visualise the simulation data with as low start-up cost as possible, while at the same time making sure that the DMC has enough information for the visualisation to show the next few frames.

One approach would be to simply parse the simulation data line-by-line in real time, keeping track of the current time of the trace file and comparing this to the timeline in the application. The start-up cost would be non-existant and it would give the user a sense of real-time simulation. However,

this method is made impractical due to the fact that events in a trace file are not grouped. (e.g. a packet transmission might span thousands of lines and then be dropped.)

Another approach is to introduce a *look-ahead* when parsing the simulation trace file, to model the internal network state ahead of the visualisation timeline [45]. At the cost of a little start-up delay, to read a certain amount of simulation data ahead, one can visualise events when they occur, since the tool has an overview of the network model a certain time ahead [45]. A question then arises, how long should this look-ahead be?

We can get a possible solution for this answer by looking at previous work. Huginn uses a solution, where the look-ahead is defined as the upper time extent of the lowest layer transmission in the simulation, the Media Access Control (MAC) layer [45]. Scheuermann defines t_s as the current visualisation time and L as the look ahead, "a span of time longer than the longest MAC layer transmission in the trace file" [45], Section 5.1. So the parser in Huginn is always $t_s + L$ ahead in the trace file, compared to the visualisation [45]. It is unclear if L is pre-computed by parsing the trace file (thus finding the longest MAC transmission in advance, prior to visualisation) or L is initially set and updated during parsing. Regardless, we decided to use a similar approach to Huginn in IBSimVis, as line-by-line reading is impractical and can make the implementations of the DMC and the VC complex. L in our Parser can either be set by the user or set intially by the program, then updated during parsing when it finds a complete InfiniBand packet transmission taking more than L time. IBA packets do not carry information about timeout in their header [16], otherwise we could use this as an upper bound of L. The challenge of grouping trace file lines into events to be visualised, is covered next.

4.3.1.3 Event correlation

After the Parser component has made sure that relevant simulation events have been parsed, one needs to group related simulation events into a cause-and-effect group [45]. Events in the event log are not grouped, meaning that

a message receive event might be thousands of lines after a message send event. That is, for the visualisation to be able to show the complete history of a packet, from source to destination, it needs to know which message events that correlate. The VC needs to know from which node the message was sent and at which node it was received. It also needs to know whether the message was dropped in either of the two switches in the example. We want to use the `messageId` field in the OMNeT++ event log, initiated by the BeginSend (BS) event [52]. The messageId is unique during an entire simulation run and can be tracked when it is deleted or any other message event is recorded. Another solution to this, presented by Scheuermann [45], is to compare fields in the message header, requiring the parser to read the header of IBA messages, in addition to the event trace itself. This has the disadvantage of being potentially slow, since one has to compare each message header to every other message header.

4.3.2 Topology

The Topology is the central component in the DMC and contains data structures that keeps track of all the entities and their attributes. In addition, it calculates statistical data derived from the entities and simulation. As mentioned in Section 2.2.2, entities are abstractions contained within data. These entities have a structural relationship, which can be modelled as a graph, making up the network topology.

4.3.2.1 Entities

Previously, we have mentioned what the Topology component does and how it communicates with the other components. Here, we briefly introduce the entities that are present in the Topology component, namely:

Node Models either an InfiniBand CA or switch, introduced in Section 2.1.3. Contains a list of ports and the LFT.

Port Attached to nodes, contains information about what link it is connected to.

Link Models a bi-directional physical link in an InfiniBand network, introduced in Section 2.1.3. Connected to two ports, each connected to a different node each.

Message The main method of communication between nodes. A message is produced and consumed by nodes and travel over links.

The Node, Port and Link entities are contained within static data, showing the structural relationship in an InfiniBand network. Messages are disseminated by the Parser component from dynamic data.

4.3.3 Layout

The Layout component of IBSimVis, has one requirement, based on the characteristic shown by for example NAM and OMNeT++, both employing a similar approach to layout network topologies, namely the force-directed approach. The requirement is to handle the presentation issue of a network topology, that is, how to display it in a way so that it is easy to understand. Here, we discuss design considerations of the Layout component, which impacts our implementation, described in Section 5.3.3.

The Layout component is designed to contain a set of implemented force-directed layout algorithms. It takes a graph maintained by the Topology component as input, and outputs a layout for the graph as points in 2D or 3D space for each node, by directly manipulating the position of the nodes. It follows that, if the nodes are placed, then the links between them also know where to start and where to end. Other components can invoke the Layout component when they need a layout applied to a graph. For example, it may be configured prior to initialisation and run during initialisation of IBSimVis. The Layout component may also be called by the GUI, to either apply a new layout algorithm or re-layout with the same algorithm.

As mentioned, this component has one essential task, being able to layout a network topology. To do so, we employ a force-directed layout approach, introduced in Section 2.3.4.3 and the reason for our approach was discussed in Section 2.3.5. In addition, both NAM and OMNeT++ use a similar ap-

proach [43, 52], giving a good presentation of the network, when the number of nodes and links in it is small. However, since we are going to deal with simulated networks with hundreds or thousands of nodes, we need to take into consideration the challenges introduced by Herman et. al. [31], mentioned in Section 2.3. Namely those of graph size, time complexity and predictability. We take a look at those layout challenges next, and how they relate to IBSimVis.

4.3.3.1 Topology size and time complexity

As mentioned in Section 2.3.5, force-directed layout algorithms generally take a long time to run. Some implementations may take up to $O(|V|^3)$ time to process [36]. So the larger the topology is, the longer it takes to converge to the equilibrium configuration. The reviewed force-directed layout algorithms such as the one by Fruchterman and Reingold requires at least $O(|V|^2 + |E|)$, *per iteration* [40]. The multilevel algorithm by Walshaw requires the same as the Fruchterman-Reingold algorithm, in addition to the time complexity introduce by the multilevel approach [29], although the graph coarsening step only happens once.

There is little we can do to reduce these demands for computation time, as to the best of our knowledge, there are no parallellised version of these algorithms. If there are, we leave it as future work to add parallellised force-directed layout algorithms to the Layout component. However, we can introduce visual elements that shows how far the layout algorithm has progressed, such as a progress bar. Another approach is to introduce options to the layout algorithm, by enabling the user to adjust layout algorithm parameters, reducing the computation time. For example, by reducing the cooling temperature or increasing tolerance of the multilevel force-directed placement algorithm of Walshaw, described in Section 2.3.5.2, the number of iterations needed for the graph to converge, decreases [29].

4.3.3.2 Predictability

As mentioned earlier, it is possible that a layout algorithm may produce very different layouts during two seperate layout computations on the same graph. Especially if the graph changes. For example, if a node or link goes down and the layout is recomputed, the new equilibrium configuration may offer a dramatically different visualisation of the network topology [57]. In turn, rendering the visualisation unusable, since the user might have become "lost" [57]. As mentioned earlier, another disadvantage of the force-directed approach is that the user may require to use time to re-learn any new layouts, breaking the "mental map" [32] the user had of the network topology. As an example, assume that the graph in Figure 4.3a has been created with a force-directed layout algorithm. Adding the edge (A, C) and re-applying the same layout algorithm, may create a layout such as the one in Figure 4.3b. Requiring the user to re-learn the structure and node positions.

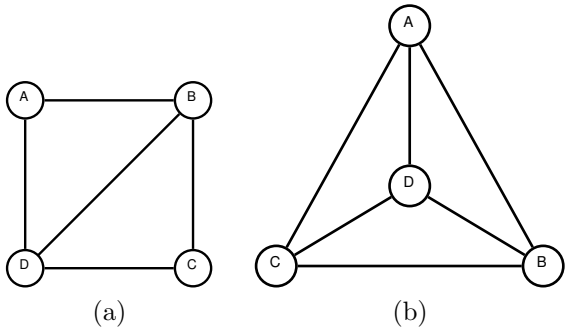


Figure 4.3: Adding an edge, (A, C) and re-applying a force-directed layout algorithm destroys the mental map. Adapted from Figure 4 in [32].

A possible solution to this issue is to introduce a layout file format, to the set of static data files, as optional input to IBSimVis. Alternatively, modify existing topology data file formats to include node positions in its definitions. Where the user the user can pre-define a layout complete with all node positions, similar to the approach used in both NAM and OMNeT++. Or let the user save a computed layout of a network in the layout file. This has two advantages; it saves start-up time by not having to run costly force-directed algorithms on a topology more than once, and the user does not

have to re-learn how the network topology looks like. The Parser component can easily be extended to read this layout file. A drawback is that the user has yet another input file to keep track of, even when using the OpenSM topology data file format, as this would not be usable by OpenSM again. Another solution is to introduce a *layout cache*, transparent to the user. The layout cache would write to the users home directory, preserving node locations after a given layout algorithm has been run, mapping the input file to the layout cache. Due to time constraints, we reserve defining layout file formats or layout caches as further work.

4.4 Visualisation Component

The Parser component is responsible for identifying the entities and attributes according to the semantics given in both static and dynamic data. The available entities we want to represent are introduced in Section 4.3.2.1. Entities may contain attributes that we want to visualise.

How to represent and display these entities and their attributes, is the task of the Visualisation Component (VC). It handles the conversion from entities to visual objects by applying various encoding mechanisms, such as the ones mentioned in Section 2.2.1. Furthermore, its subcomponents, the Graphics Engine displays the symbols (visual objects) to the user. While the GUI offers options for giving extra information about visual objects, allowing data filtering, interaction, simulation playback and statistics. The Graphics Engine and GUI in IBSimVis use high-level abstractions from third-party libraries, introduced in Chapter 5. We move on to discuss the design issues and challenges of the VC. Starting with a look at the presentation problem in information visualisation, by deciding to visualise in 2D or 3D.

4.4.1 2D or 3D?

In Section 2.2.2, we saw that one of the main problems with presentation is the lack of display space. As mentioned earlier, in IBSimVis and the reviewed related work, the view of the network topology is central to the visualisation.

Looking at the tools in related work, most use a 2D layout for the topology. With the exception of Huginn, that uses 3D but the topology itself is drawn in a 2D scenario. The third dimension used for statistical data in the form of bar charts [45]. The main drawback of a 2D layout is the amount of zooming and panning needed to navigate the topology.

To increase the available display space, 3D instead of 2D techniques have been suggested [6,21,31]. First of all, as we mentioned, 3D offers more space due to the extra dimension [31,58]. Second, our world is in 3 dimensions and it is easier for us to relate to [58]. Third, it is easy to extend force-directed layout algorithms to 3D [29, 31, 40]. Another argument to presenting the visualisation in 3D, is to offload visualisation calculations to the Graphics Processing Unit (GPU), leaving Central Processing Unit (CPU) resources free for other tasks.

There are several drawbacks when using 3D. On a display screen, we render 2D projections of 3D scenes [58]. The infinite possible viewing angles makes it hard to deal with edge crossings [58]. Although a layout algorithm does not produce edge-crossings in 3D, it is hard to avoid when viewing it. Navigating in 3D is difficult to handle in a user-friendly way, due to the six movement directions [6]. In addition, 3D tools can be harder to implement than their 2D counterparts [6].

A compromise between a 2D and 3D topology layout could be done, giving the user an option to switch between them. For example, by "flattening" the topology, by setting the z-value to 0, placing nodes according to $position(x, y, 0)$. Or by providing 2D and 3D implementations of layout algorithms.

Due to the advantages mentioned above and the fact that 3D visualisations tend to be visually exciting [6], we have decided to make IBSimVis able to display a network topology in 3D, with the ability to switch to flattened 2D layouts if the user wants to. As Fruchterman and Reingold put it: "In summary, a three-dimensional layout stands or falls according to the preference and expectations of the user." [40], page 1156. In the end, 3D is the default and preferred view in IBSimVis. Note that we do not extensively cover the background and challenges around 3D rendering and programming,

even though we do use and refer to some basic terminology. For more information about those topics, we refer to the the books by Foley et. al. [59], Akenine-Moller et. al. [60] and Schreiner et.al. [48].

4.4.2 Choosing the background

The network topology is central to our visualisation. To be able to discern glyphs in the network topology visualisation, some care must be taken to choose a background that contrasts with the glyphs to make them preattentively distinguished. In general there are two main approaches.

The first is a static coloured background colour, such as white, seen in iNSpect, OMNeT++ and NAM. The choice of background colour is based on personal preference. The only criteria is that glyphs encoded with colour, needs to contrast well with the background [21], for this black or white are suggested [21].

Another approach is to have a texture as a background, given that the texture is able to contrast colours and possibly other textures well. An advantage is that it could also help give context to the visualisation. For example, OMNeT++ and iNSpect allows a custom texture background [49, 52]. Huginn has a texture background of a mountain range, as seen in Figure 3.3. Textures can be helpful, giving context to the visualiation. A texture of a mountain range might give context to a search-and-rescue scenario where nodes transmit to eachother or a desert texture might give context to a military scenario, simulating communication between military units.

We decided to go with a black background, because it contrasts well with the colour set suggested by Ware [21]. We believe that the alternative, a textured background in IBSimVis is not applicable to our simulation setting or scenario.

4.4.3 Supported encoding mechanisms

As mentioned in Section 4.4.1, we are going for a 3D approach. The actual transformation of entities to graphical primitives is handled by the Graphics Engine, typically using high-level abstractions, calling a low-level graphics

API such as OpenGL or Direct3D. It is the job of the VC, to decide what the glyphs look like. Here, we briefly mention the supported encoding mechanisms the VC is going to handle.

IBSimVis supports the most common encoding mechanisms mentioned in Section 2.2.1. Namely, position, size, colour, texture, shape and orientation. The reason for this is that they are easy to implement and their visual properties make them easy to distinguish from each other [21]. These encoding mechanisms, are the basic building blocks of our visualisation. However, the choice of which encoding mechanism to apply to which entity or values connected to entities is open to debate. We discuss this in the next section.

4.4.4 Entity representations

So far we have introduced the entities that exist in the DMC, and what encoding mechanisms that are supported in the VC. Here, we discuss what encoding mechanisms we want to apply to a given entity, to create its glyph. As mentioned in Section 2.2.1, it is desirable for glyphs to be preattentively distinct from each other. Thus, we have decided to create one basic representation per entity. These representations might change depending on what type of view the user decides to choose. The entities are covered in Section 4.3.2.

Nodes can be one of two types: CA or Switch. These are perceived as different, so we need to represent them differently. There are several ways to do this. One, is to use colour, for example green for switch and red for CA. Another, is to use different shapes, such as a cube for switch and sphere for CA. In literature, we saw a similar approaches. Dally showed switches as squares and terminals as circles [12]. Kurkowski et. al. used spheres for nodes and coloured them according to state [49]. And Huginn represented nodes as cones [45]. Another approach is to use textures to show the difference. A switch might have a texture that looks like a switch and a host a texture that looks like for example a normal PC. We decided to join the approaches by Dally and Kurkowski et. al., in addition to extending the representations to 3D, by representing CAs as coloured spheres and switches as coloured

cubes. Mainly because these encoding mechanisms are simple to implement. In addition, texture encoding needs texture files, requiring creative skills that we do not posess.

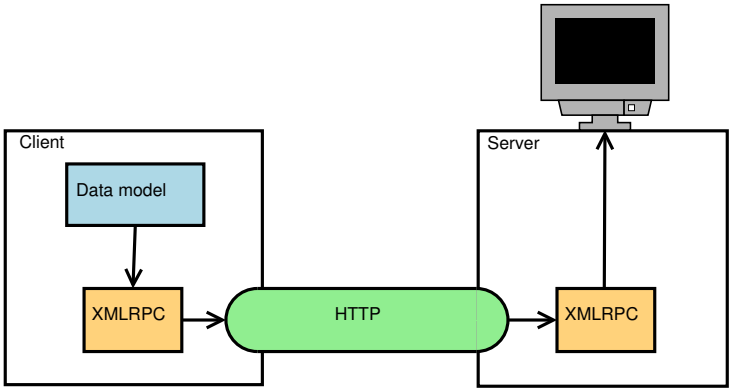
Ports can be represented as small, coloured shapes attached to a Node. Alternatively, we can omit them from the visualisation, due to their small size. We have no effective method of distinguishing them from each other, since the number of ports on a Node can be high. As mentioned in Section 2.1.3, switches can have max 256 ports. Not only would a switch look cluttered, if we represented them as shapes and attached them to the Nodes, placing them around a cube in a smart way, is a challenge in itself. Thus, we have decided to omit representing Ports in the visualisation, instead adding them to an information table in the GUI, when the user selects a Node.

Since we use force-directed placement algorithms, that utilise straight-line drawings, we want to represent Links as such. We also want to be able to encode links with thickness (size) and colour. Thickness may denote the number of multiple edges between two switches or links that are prone to congestion. We also want to be able to both connect and disconnect links, as the simulation may change the network topology.

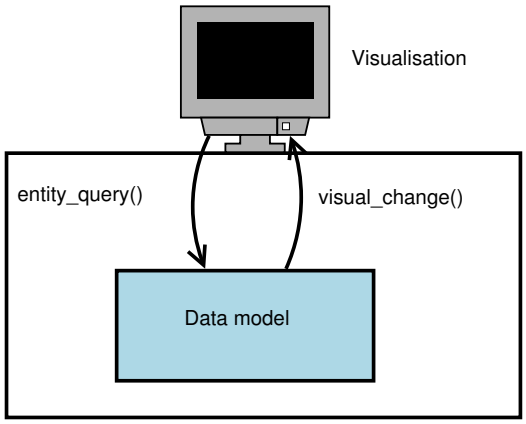
We want to represent a Message as a cube travelling over links from a source to their destination, this is similar to the Packet building block in NAM. We want packets to be encoded with colour, depending on what type of traffic they represent. (e.g. red for MADs and blue for normal traffic) In addition, they should be able to move along the links, passing through nodes.

4.4.5 Communication with DMC

To be able to create symbols and show changes, one needs a method for communicating between the DMC and VC. The DMC needs to be able to inform the visualisation of what entities look like and if they change during simulation playback. In addition, the VC needs access to entities and their attributes. (e.g. a user selecting a node, wants information about that node) We have observed two main approaches to inter-communication between the data model and the visualisation.



(a) Client-server approach in UbiGraph [61] using XMLRPC, showing communication between a data model and its visualisation



(b) Direct function call approach used by NAM, iNSpect and OMNeT++, showing direct function calls between a data model and its visualisation.

Figure 4.4: Two approaches when communicating between a data model and its visualisation. One using a client-server approach and the other using direct function calls.

First, the client-server model used by a graph visualisation tool named UbiGraph [61]. The server offers visualisation as a service to the client, or several clients. The communication between the server and client, utilises Extensible Markup Language Remote Procedure Call (XMLRPC) over Hypertext Transfer Protocol (HTTP) to push changes to the visualisation [61], as illustrated in Figure 4.4a. The XMLRPC call from the client to the server, also contains information about nodes or edges, such as a label for their name,

so the server does not have to query about this when the user interacts with the visualisation. There are three advantages to this approach. One is that the client can be distributed on several computers. Second, both the client and the server are loosely coupled, and thus either can be replaced as long as the communication semantics stay the same. Third, static entities in the data model is contained in the visualisation, querying for entity attributes is not necessary. A disadvantage of the approach used in UbiGraph is the potential overhead in communication on the server-side, due to parsing Extensible Markup Language (XML) [61]. Limiting the client Application Programming Interface (API)-calls to 1-2 thousand, compared to $10^5 - 10^6$ API-calls when calling the visualisation directly [61]. Small changes to the visualisation might take time, due to parsing delay, especially if the communication bandwidth is bad and there are alot of changes that happens rapidly. A similar approach is used in Huginn, but the communication medium is different, with Huginn's EE only pushing visualisation changes to its VE through a UNIX pipe [45].

The other approach, used by NAM, iNSpect and OMNeT++ is a tight integration between data model and the visualiation, relying on direct function calls in software. This approach is illustrated in Figure 4.4b. This has the advantage that communication overhead is almost non-existent. Changes are immediately displayed. A disadvantage is, that the tight coupling between the components makes the system less flexible. We decided to go for the latter approach, since implementation is easier and the potential amount of events in the data model that needs to be pushed to the VC might be substantial. Mainly because our problem domain is concerned with simulating high-speed InfiniBand networks. Although, the advantages to the former approach are very interesting, we leave this as a future extension to IBSimVis.

4.4.6 Presentation and interaction techniques

Here, we briefly mention the presentation and interaction techniques used in IBSimVis and briefly discuss some design considerations. For the implementation details, we refer to Chapter 5.

In IBSimVis, we use geometric zoom. However, it would be interesting to explore the use of semantic zoom. Zooming in with semantic zoom, can for example show the internals of switch entities, with buffers filling with messages, going through ports or even simulation aspects of even lower granularity. However we reserve this as further work, since the use of semantic zoom requires alot more design and implementation considerations, which is outside the scope of this thesis.

As mentioned in Section 4.1.2, using 3D gives additional challenges to presentation and interaction. To be able to view more than a 2D presentation of a 3D model, we need to add the ability to rotate around the model. We argue that rotation, coupled with zooming and panning is enough to navigate the 3D environment.

In IBSimVis, we want users to be able to select entities in the visualisation, to get more information about them. This is similar to the tools in related work, where nodes could be selected and one would get information about their internal state. We do this by introducing the Entity Selection Query. Nodes and links, the main glyphs in our visualisation, can be clicked on by the user and information about them queried for. Selecting nodes in 3D space poses several challenges, which we tackle in our implementation.

Features deemed important in any visualisation tool, is not only being able to view data from different angles, but also to choose whether to do so. As mentioned in Section 2.2.2, data filtering is of utmost importance to any visualisation tool. As an example, Huginn has an entire pre-configuration step using a flowchart editor for filtering data. In addition, the more general graph visualisation tools such as Cytoscape and Gephi, have data filtering as a core feature [8, 9]. Data filtering is important in IBSimVis, since we want to visualise network topologies, where nodes and links occlude each other. As Card et. al. put it: "Visual representations of generalized graphs of even modest size tend to look like a ball of tangled string" [6], page 187. There is little we can do to untangle the ball of string, as the layout algorithm we have applied already has attempted to do so. However, we can make patterns come out more clearly by giving the user the option to filter data. In IBSimVis, the user is able to filter data using Dynamic Queries, by for

example dragging sliders in the GUI or tweaking number filters.

A common characteristic among the tools in related work, is the ability to do simulation playback. The interactive element of this, is commonly shown as a timeline slider. Allowing the user to manipulate the timeline slider, enables the user to skip to an arbitrary point in simulation time. However, a timeline slider poses a number of challenges. Not only, because the time granularity of packet transmissions, in InfiniBand networks can be as low as nanoseconds. We already presented the design of the Parser subcomponent in Section 4.3.1. Assuming the DMC and VC has enough data to visualise given time slices, there are more challenges. For example, playing forwards in a linear fashion. Alternatively a FlexTime approach used in Huginn could be used, where simulation playback increases, based on the numbers of events occuring ahead in time. Consequently, slowing down simulation playback when there are numerous events. Skipping to an arbitrary point in time also poses several more issues, in regards to parsing strategies. In IBSimVis, we want to implement a simple timeline to playback simulations in a linear fashion, leaving other playback features as further work. For strategies to handle problems related to playback of simulations, we refer to [45], Section 5.

4.4.7 Views

In this section, we present the views that we want present in IBSimVis. While the number of different views can be many, we have chosen three based on user needs. These can be considered important when considering the performance of for example, routing algorithms. Only one of the views described below, can only be used to analyse static data. The other two, can be used when viewing simulation when it changes over time.

4.4.7.1 Normal

The normal view is shown by applying encoding mechanisms, with default values to the entities.

4.4.7.2 Routing

The routing view can be selected if the user wants to analyse routes between two CAs. In the GUI, it shows details about each hop in the route. And in the visualisation, the route between the CAs is shown by colouring the links green. To limit the effects of occlusion in 3D, only the links in the route and the nodes involved in the hops, in addition to their closest neighbours are shown. The rest are hidden from view.

The routing view can for example, be used to compare two routing algorithms, by showing the difference in the route. In addition, the NumPaths view can be applied, showing that paths along the route may be potential choke-points.

4.4.7.3 NumPaths

The NumPaths view is the most important view in IBSimVis. The purpose of this view is to show the path distribution in a network topology, which is calculated by the Topology subcomponent in the DMC. This feature we consider important, because it is one of the basic tools used when researchers evaluate routing algorithms.

The NumPaths view is created by encoding all node and link entities in different colours, based on the number of paths that go through each link and node. For example, considering a regular fat-tree topology, a view such as NumPaths can show if a given routing algorithm makes the topology unbalanced, by encoding the node and links with the most paths in a deep red colour. This can draw the attention of the user so that he can address the issue or possibly find bugs in the routing algorithm.

However, two issues arise. One is what intervals the number of paths should be, assuming the minimum number of paths over a link is $link_{min}$ and the maximum, $link_{max}$. A simple solution to this issue is to take $link_{max}$ and divide by a constant, C. This way, we each interval is $link_{max}/C$. A drawback of this solution is that if most links in a network topology has path values closer to $link_{min}$ and there is one really large $link_{max}$, most of the links would be coloured in the same colour. And other important links

except the one that is $link_{max}$ might be overseen. Another solution is to give the user the ability to set his own path intervals, giving good customisability and flexibility. However, this could be a tedious process for the user. We opted for a compromise, to show the former approach as the default, with a C set to 5, with the user able to optionally tweak the interval values. In the future, we would like the user to be able to also configure C.

The other issue is how to colour nodes and links according to the number of paths that go over them. One method is to simply hardcode the colours according to each interval. The other, is to allow the user to choose his own colour for intervals using a colour pallette, mentioned in Section 2.2.3. This issue is also linked to the number of intervals C, considering each interval is to be assigned its own colour. Ware recommends that the colour set in a visualisation tool, to between five and ten, because "only a small amount of colours can be rapidly perceived" [21], page 125. We choose the colour pallette approach, since it gives the user customisation options and since such a feature is easily available in most GUI libraries.

4.5 Summary

In this chapter, we have discussed design challenges in developing a prototype visualisation tool for OpenSM topologies, routing data and OMNeT++ simulation traces, named IBSimVis. Inspired by tools in related work, we have introduced the simple design of the stand-alone visualisation application. The design is limited to visualising static network topologies and packet traces from simulation data. It consists of two main components, the DMC and VC. The main components communicate directly together, using function calls for minimal communication overhead. Furthermore, the following subcomponents have been defined, each handling a requirement introduced in Section 4.2. The Parser component handling parsing of static and dynamic data. The Layout component handling how to layout a network topology. The Graphics Engine component handling presentation and interaction of a network topology in 2D and 3D. And the GUI component handling data filtering, simulation playback and view change.

The Parser component in IBSimVis processes OpenSM-defined network topologies and routing data. It also uses a look-ahead and event correlation strategy to parse simulation data. The Topology component contains four main entities: Node, Port, Link and Message. Each representing a different part of the static and dynamic data input to IBSimVis. IBSimVis features an interactive 3D topology view on a black background, using a force-directed layout algorithm to position nodes. There is an option to flatten the topology into 2D. The VC can encode entities using the shape, colour, size and texture encoding mechanisms. Interaction techniques included is geometric zoom, rotation, panning, Entity Selection Queries for selecting entities in the 3D view and Dynamic Queries to filter data. IBSimVis features a timeline slider in the GUI, visualising simulation data changes in linear time. In addition, there are three defined views in the GUI: Normal, NumPaths and Routing view. These enable a change of context in the 3D visualisation. The Normal view is the default view, showing entity representations with default values. The Routing view shows the path from a CA to another CA, hiding the rest of the topology, except the immediate neighbour links and nodes to each hop. The NumPaths view shows the path distribution of the network topology.

In the next chapter, we present the implementation details of IBSimVis. We realise the features summarised above and attempt to tackle the design challenges introduced in this chapter.

Chapter 5

Implementation

In this chapter, we present the implementation details of IBSimVis. We go into detail of how the components we discussed in Chapter 4 were implemented and how they communicate.

Due to time constraints, we had to omit the simulation aspect when implementing IBSimVis. Thus, the current prototype is used for visualising static properties of OpenSM-defined InfiniBand subnets, such as network topology and routing tables. The reason why we had to narrow our implementation down, was the implementation complexity of the 3D and GUI programming, as we had no prior experience doing this. The ability to visualise simulations is reserved for future work. We believe that our design can aid the implementation of simulation data visualisation, if done in the future.

The IBSimVis application has gone through two major prototype iterations. In the first prototype we focused on the data structures and layout mechanisms, while using a simple 3D interface. In the second prototype, we focused on the visualisation parts, considering various third-party libraries for handling the GUI and Graphics Engine components in the VC. The second iteration required a tight coupling between the VC and DMC, due to the design of the third-party GUI library we used. In this chapter, we sometimes refer to either of the two prototype iterations. This is to show that IBSimVis has evolved during our thesis work. We have also focused on implementation on the GNU/Linux platform, reserving other platforms as further work. This

should be no problem, as all utilised third-party libraries are cross-platform compatible.

We start this chapter by discussing our choice of implementation language in Section 5.1 and differences from our original design in Section 5.2. Then move on to discuss implementation details of each component outlined in the design, in turn. We discuss the DMC implementation in Section 5.3, the VC implementation in Section 5.4 and GUI implementation in Section 5.5. When we discuss component implementation, we reflect over choice of third-party libraries, data structures, program flow and implementation issues, with some relevant pseudocode or C++ code at select places. Towards the end, we briefly mention some license considerations. Lastly, we provide a summary of this chapter.

5.1 Programming language

We chose to implement our visualisation tool in the object-oriented programming language C++ [62]. C++ is the most actively used programming language for developing 3D games and applications. This is evident as 261 of 321 (81.3%) currently listed graphics engines at [63], are implemented in C/C++ or have bindings to it. This is most relevant since we want to explore the use of 3D to encode data, as discussed in Section 4.4.1.

In this thesis, we use the Standard Template Library (STL) in C++ and other C++ standard library features and terminology. In addition to features in the future C++ standard, C++0x already present in the GNU's Not Unix (GNU) C++ compiler, gcc. For example, the `auto` keyword used for type inference. We also refer to and use parts of the Boost C++ library, which is commonly used for C++ applications [64]. Furthermore we refer to namespaces such as `std`, referring to the C++ standard library. In addition, we also mention some object-oriented programming terminology such as classes, objects, member functions and inheritance.

5.2 Differences from design

In Section 4.2, we discussed the design overview for IBSimVis. We discussed the division into two main components, the Data Model Component (DMC) and the Visualisation Component (VC), where these two communicate with each other through direct function calls. One of the main changes from the design, is how the GUI subcomponent in the VC communicates with the rest of the application. In our original design, our intent was to make the GUI component communicate with the DMC through functions in the VC. However, the GUI library we chose for our implementation changed this. Therefore, we present a new component design overview in Figure 5.1. The original design overview is shown in Figure 4.2. This time, the GUI application wraps around both the main component, itself being a supercomponent in IBSimVis, controlling the other components. Another difference is that we have implemented the VC inside a GUI widget. A widget being a basic visual building block in a GUI system. Thus, GUI library calls are prevalent in almost all components, since we now communicate directly with the GUI. We discuss why the design change was necessary in Section 5.5.3 and the GUI library we opted to use, wxWidgets in Section 5.5.1.

5.3 Data Model Component implementation

In this section, we present the implementation of the Parser, Topology and Layout components, that we discussed in the design.

5.3.1 Parser

As stated at the start of this chapter, we do not visualise simulation data. So we do not describe implementation details of parsing simulation data which was listed as a task in our Parser design, in Section 4.3.1. Instead, we have implemented parsing topology and routing data.

The input comes from the command-line parser in the GUI component, explained in Section 5.5.2. The topology data file, with `lst` extension is a

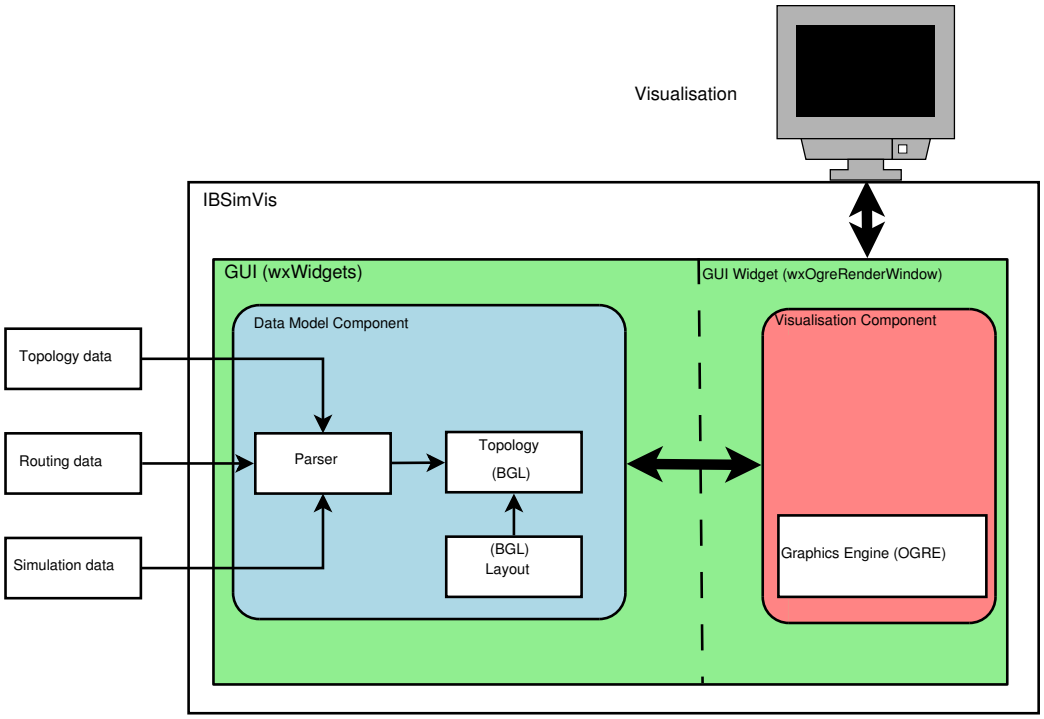


Figure 5.1: Revised component design overview in IBSimVis, showing components and inter-component communication. Note that the GUI component now is a supercomponent controlling the main components.

list of all the nodes, ports and links in the fabric. The routing data file, with `fdbs` extension, contains Unicast Linear Forwarding Tables (LFTs) of the switches in the fabric. Because both file formats lack documentation, we had to read the source code of both OpenSM and IBMgtSim to understand their semantics.

As mentioned in Section 4.3.1, parsing topology- and routing- data is straight-forward. Parsing itself is done by defining regular expression grammar [55] using the `boost::regex` library and the file reading capabilities in C++ `<iostream>`. Another approach would be the input data to a scanner tool such as flex [65] to recognise lexical patterns and for tokenisation. Then provide the tokens to a parser generator tool such as GNU Bison [66], together with a syntax grammar. However, we opted against this approach, since the grammar in the two file formats we are parsing is simple. Writing syntax grammar for Bison and lexical definitions for flex would require us to

learn the tools and then write the required configurations needed. For more advanced parsing strategies and an overview of the parsing craft, we refer to the book by Grune et. al. [55].

After the main application has gathered the input and options for IBSimVis, the parser is intialised with the input files mentioned above, as parameters. The Parser component is implemented as a class named `TopologyParser`, taking a `Topology` pointer as a parameter in its constructor, to make sure entity creation is fast, as discussed in the design. Parsing the topology data file is implemented in a class member function called `parseLstFile` and parsing routing data in a class member function called `parseFdbsFile`.

The `TopologyParser` class confirms whether the input files are of the correct extension. That is, the topology data files should have the extension `lst` and the routing data file the extension `fdbs`. `TopologyParser` assumes that if the extensions are correct, then the file format is correct. `TopologyParser` does not do any format verification checks, since such checks can be tedious to implement. We assume that the user inputs the correct file formats when starting IBSimVis. When this is set up, the main thread in the GUI component, either calls `parseLstFile` or `parseFdbsFile` to parse each file format, respectively. The Parser component fails if the file reader reports a failure or the line fails to match the regular expressions in either parsing function. We describe their formats and parsing implementation in Section 5.3.1.1 and Section 5.3.1.2, respectively.

5.3.1.1 The topology data file

The topology data file is logged after OpenSM has discovered a subnet, using IBMgtSim to simulate the existence of a cluster. The format is a list of lines, showing nodes, ports and links in an InfiniBand fabric. Each line is an unidirectional relationship between the ports of two nodes. The nodes can either be a switch and a CA or a switch and another switch. Table 5.2 describes the fields identifiers, their descriptions and integer base. Table 5.1 shows the three different available node-types. The node type is defined in the first token of the uni-directional line, as shown in Listing 5.2.

Table 5.1: Node type and description

Type	Description
SW	Switch
CA	Channel Adapter
CA-SM	Channel Adapter acting as Subnet Manager

Table 5.2: Topology data file field descriptions. Showing the field identifier, their semantic and what integer base they are.

Field identifier	Description	Base
Ports	Number of ports on the node	16
SystemGUID	GUID of system	16
NodeGUID	GUID of node	16
PortGUID	GUID of port	16
VenID	ID of vendor	16
DevID	Vendor-assigned device ID	16
Rev	Revision ID	16
LID	Local Identifier of the port	16
PN	Port number on the node	16

Two lines in a sample topology data file is shown in Listing 5.2. The first line, shows a CA that is acting as a SM with GUID 0x0002c90000000001 and is connected through port 1 to a switch with GUID 0x0002c90000000015 at port 1. The second line shows the other direction. Together these two lines make up a bi-directional link. In Listing 5.1 we show the regular expressions that we used. Each text string containing digits or alphanumerical characters following the field identifier in Table 5.2 is matched to an attribute in a `Node` entity, with the `portLine` itself identifying it. Note that some parts of the regular expression grammar are *greedy* when using the wildcard signs "*", meaning that they match whatever is inside the delimiters [55].

Listing 5.1: Regular expressions used to extract field data in a topology data file. Each line in a topology data file, consists of a portLine, blank space, a portLine and an endLine.

```
string portLine = "\\{\\s+(.*)\\s+Ports:(\\d+)\\s+SystemGUID
    :([0-9a-zA-Z]+)\\s+NodeGUID:([0-9a-zA-Z]+)\\s+PortGUID
    :([0-9a-zA-Z]+)\\s+VenID:([0-9a-zA-Z]+)\\s+DevID:([0-9a-zA
    -Z]+)\\s+Rev:([0-9a-zA-Z]+)\\s+\\{(.*)\\}\\s+LID:([0-9a-zA
    -Z]+)\\s+PN:([0-9a-zA-Z]+)\\s+\\}.*";
string endLine = "\\s+PHY=(.*)\\s+LOG=([A-Z]+)\\s+SPD=(.*)";
boost::regex regex(portLine+"\\s+"+portLine+endLine);
```

Once a `Node` entity at either end has been identified, we proceed to look up both their GUID in the Topology component. If they do not exist, they are created. Only if both exist, either by `nodeMap` lookup or creation, two `Port` entities are created, with the portGUID and port number. The `Port` entities are then connected together, via a `Link` entity. In the implementation, these are are pointers.

5.3.1.2 The routing data file

The routing data file is logged by OpenSM after it has first discovered a subnet, and after a routing engine has been applied to the subnet. It consists of a list of LFTs of all the switches in the subnet. Listing 5.3 shows an LFT table for a switch. Each forwarding table in the routing data file is divided in two parts; an initialisation line followed by a list of what we

Listing 5.2: Sample topology data file, showing a bi-directional connection between two ports.

```
{ CA-SM Ports:02 SystemGUID:0002c90000000001 NodeGUID:0002
   c90000000001
  PortGUID:0002c90000000002 VenID:000000 DevID:5A44 Rev
     :00000000 {H-1 HCA-1
    (Mellanox HCA)} LID:0001 PN:01 } { SW Ports:18 SystemGUID
       :0002c90000000015
  NodeGUID:0002c90000000015 PortGUID:0002c90000000015 VenID
     :00000000
  DevID:B9240000 Rev:00000000 {S-0Infiniscale-III Mellanox
     Technologies}
  LID:0006 PN:01 } PHY=4x LOG=ACT SPD=2.5

{ SW Ports:18 SystemGUID:0002c90000000015 NodeGUID:0002
   c90000000015
PortGUID:0002c90000000015 VenID:000000 DevID:B924 Rev
   :00000000 {S-0
Infiniscale-III Mellanox Technologies} LID:0006 PN:01 } { CA-
   SM Ports:02
SystemGUID:0002c90000000001 NodeGUID:0002c90000000001
   PortGUID:0002c90000000002
VenID:00000000 DevID:5A440000 Rev:00000000 {H-1 HCA-1 (
   Mellanox HCA)} LID:0001
PN:01 } PHY=4x LOG=ACT SPD=2.5
```

refer to as forwarding entries, to every LID in the subnet. The initialisation line identifies which switch the forwarding table belongs to. The string `_-_osm_ucast_mgr_dump_ucast_routes` prior to the switch GUID is skipped. Each forwarding entry consists of the destination LID, outgoing switch port, number of hops until destination and optimal flag. The optimal flag was originally used as an indicator whether the route was the shortest path in a fat tree, but is rarely used. Note that we do not parse the hop and optimal flag fields, as they are not used in our implementation, we reserve this as future work. Referring to Listing 5.3, if a message wants to reach LID 0x0008, it has to go through outgoing port 002 on the switch. After the switch GUID has been parsed, the Parser component looks it up in the Topology component. If the switch exists as a `Node`, the LID and port number is matched using regular expressions and inserted into the LFT of the `Node`.

Listing 5.3: Sample fdbs file, showing the LFT for one switch.

```
__osm_ucast_mgr_dump_ucast_routes: Switch 0x0002c90000000015
LID : Port : Hops : Optimal
0x0001 : 001 : 01 : yes
0x0002 : 002 : 02 : yes
0x0003 : 002 : 03 : yes
0x0004 : 002 : 04 : yes
0x0005 : 002 : 05 : yes
0x0006 : 000 : 00 : yes
0x0007 : 002 : 01 : yes
0x0008 : 002 : 02 : yes
0x0009 : 002 : 03 : yes
0x000A : 002 : 04 : yes
```

5.3.1.3 Implementation issues

One of the first issues we encountered, was how to turn a hexadecimal string into an unsigned 32bit or 64bit integer. We could have parsed each character in the hex string, leading to a complicated implementation. Instead, we found the solution on StackOverFlow, using `std::hex` and stream insertion operators on the string [67].

Another issue we encountered, was the non-uniform way port numbers in the topology data and routing data files formats were used. In the topology data file format, a port number is in hexadecimal. In the routing data file format, the port number is in decimal. This quickly led to problems in the parsing, when the Parser component encountered letters in the "PN:" field. Leading to a matching failure and Parser component shutdown.

5.3.2 Topology

The Topology component is the main datastructure in IBSimVis. As mentioned in the design, it keeps track of all entities (objects), their data and their relations. It has convenience functions to retrieve information about topology, routing, construction and modification of the network topology. Note that when we refer to `classes` contained in the Topology component, they are essentially the same as the entities mentioned in the background and design. The same goes for member variables and entity attributes.

The Topology component is implemented in a set of classes: `Topology`, `Node`, `Link` and `Port`. The `Topology`, which is the main class in the Topology component, keeps track of two main data structures. One is the network topology containing all the nodes the other is the graph implementation used by the Layout component. The network topology data structure is implemented using the STL key-value associative container `std::map`, with the local member variable called `nodeMap`. The node GUID, a 64bit unsigned integer is the key and the value is the pointer to a `Node`. This allows for $O(\log n)$ lookup and insertion. Lookup can also be done using the `std::map operator[]`. But it also inserts key to the map if key is not found, which can cause false nodes being inserted. An alternative would be to hash the GUIDs and use a hashmap container instead for constant lookup and insertion. However, hashmaps are as of yet not available in the C++ standard library. Note that the `std::map` must not be confused for a hashmap, as its implementation is actually a self-balancing binary search tree [62]. The `nodeMap` itself is primarily used by the `Topology` class, but it provides public service functions such as `getNodeFromGUID` for other components to call it, typically used by the GUI to lookup nodes and by VC and layout component to find node positions.

The `Node` class, keeps track of a `std::vector` container, with pointers to `Port` objects, it is a local member variable named `ports`. At `Node` construction (done in the Parser component), it initialises the number of ports as indicated in the Ports field in the topology data file. For example, if a switch has 24 ports then 24 ports are allocated, although not all may be plugged to another port on a different node. The advantage of this is that we can address a `Port`, using the `std::vector operator[]` with the port number, minus one, in constant time (compared to for example $O(\log n)$ in a `std::map` container). The disadvantage is that we do allocate more ports than necessary, using more memory. However, we need a fast lookup on ports since we use it frequently when for example calculating the path distribution of a network topology. The `Node` class, when it is of type switch keeps track of a Unicast Linear Forwarding Table (LFT). The LFT is stored as a `std::map` with a LID as key and port number as value, named `LFT`. It

could also be implemented as a `std::vector`, for constant access with the LID as the index and port number as the value. But LIDs might not start at 0, thus requiring to allocate all indexes up to the highest LID address that exists in the fabric. Since the LFT usually contains the LID and port number pairs of *all* the LIDs in the entire fabric. Node positions are stored as an `Ogre::Vector3` in a member variable called `pos`, allowing for 3D positions in (x, y, z) coordinates.

The `Port` class keeps track of what `Link` it is connected to on the other side, using a pointer. The `Link` class knows which two `Ports` it is connected to. In addition, both the `Node` and `Link` classes have a pointer to an OGRE `ManualObject`, their entity representation, which we cover later in this chapter. Note that all the entities (classes) mentioned here all have additional attributes that are not relevant to list here. For more information, we refer to the Doxygen generated API documentation at ((TBA)). As mentioned in the design, the Topology component calculates the statistical data in IBSimVis. We cover this functionality in the next section, where we look at how we calculated path distribution.

5.3.2.1 Calculating path distribution

To be able to show the path distribution over the network topology, we first need the topology itself and the routing data tables, which has been done by the Parser component. The `Topology` class has a `routeAll` function to calculate path distribution, which is called once per topology, typically at startup of IBSimVis. Each `Node` and `Port` entity has a member variable, `numPaths` to keep track of the amount of paths going over them. This is an unsigned 32bit integer. As shown in Pseudocode 5.1, we route all CAs to all CAs. By first iterating through the CAs in `nodeMap`, as the source CA. Then an inner loop, iterating through all potential destination CAs, except itself. We only route from a given source CA to a destination CA once. This is shown at line 5 in Pseudocode 5.1, where we route to the first port LID of the destination node.

The `route` function is shown at line 11 in Pseudocode 5.1. Here, we route

Pseudocode 5.1 The routeAll and route functions in pseudocode. Used for calculating path distribution of a network topology.

```
1: function routeAll
2:     for all source_CA ∈ V do
3:         for all destination_CA ∈ V do
4:             if source_CA ≠ destination_CA then
5:                 ROUTE(n_src, n_dst←Port1←lid);
6:             end if
7:         end for
8:     end for
9: end function
10:
11: function route(source, destinationLID)
12:     cur = source←Port1←LocalNode;
13:     while cur ≠ CA and loop < 20 do
14:         loop = loop + 1;
15:         { increment numPaths in cur};
16:         nextPort = cur←LFT[destinationLID];
17:         if cur = NULL and nextPort = NULL then
18:             break;
19:         end if
20:         { increment numPaths in nextPort};
21:         cur = nextPort←RemoteNode;
22:     end while
23: end function
```

the source node, `src` toward the destination LID, `dstLid`. From lines 13 to 21, we route toward the destination, by looking up in the LFT of each switch, visiting each outgoing port in turn, until we reach the destination. There exists a loop counter to avoid the function spinning, in case the routing table contains a cycle. Each visitation of either a `Node` or `Port` increments their `numPaths` member variable by one. At the end of the `routeAll` function, not shown in the pseudocode, we record statistics needed for the GUI, such as the minimum number of paths over a port and node and the maximum number of paths over a port and node. We also record `numPaths` in the `Link` entity, by summing up the `numPaths` of the two `Port` entities it is connected to. It is clear from the algorithm in Pseudocode 5.1, that the time complexity is $O(n^2)$. The function does however, have a potential for parallelisation by routing toward several destinations at once, but it would require locks on the `numPaths` variables in both `Port` and `Node`, to avoid race conditions. We leave parallelisation as future work.

5.3.3 Layout

In this section, we describe how we implemented the Layout component, which handles how to layout a network topology. Earlier we mentioned that the input to a layout algorithm, is a graph. The network topology data structure used when we described the Topology component models a graph, but it is tedious to use. Mainly because to get to a node, we first need to go through the port. Then the link. Then the port on the other side of the link and then the local node on the remote port. Therefore, we decided to use a graph library that provides an effective and re-usable graph structure implementation. The choice stood between two popular graph libraries. The Boost Graph Library (BGL) [68] and the Library for Efficient Modeling and Optimization in Networks (LEMON) [69].

The BGL is part of the Boost C++ library and provides highly extendable generic interfaces for storing and traversing graph structures [68]. LEMON is a library that focuses on effective implementations of general data structures, with a focus on graph data structures. The advantage of the BGL is that

we have used other parts of that library suite in many other parts of our implementation, so we do not require a dependency to another library. It is also a highly configurable and fast. The disadvantage comes at the cost of syntax complexity. We found that the BGL was quite hard to use, since we find generic programming (templates) in C++ to be hard. We found that the advantage of LEMON is its simple syntax and semantics. In addition, it claims to be faster than BGL in certain algorithm implementations, such as Dijkstra's algorithm [70].

In our first prototype, we implemented the layout algorithm using the BGL, but found it hard to use. Later, when implementing our second prototype, we tried a LEMON implementation. However, we discarded this implementation due to performance issues with our implementation of the layout algorithm. Hence, we continued using the BGL. We discuss this when we evaluate IBSimVis, in Chapter 6.

5.3.3.1 Single-level force-directed placement algorithm implementation

As discussed in the design, we decided to implemented the multi-level force-directed placement of Walshaw, in a class called `MLFDP` (standing for Multi-Level Force-Directed Placement). The `MLFDP` class inherits from a `LayoutAlgorithm` class, which contains general features for layout algorithms and functions for both laying out nodes randomly in 3D and post-flatting the layout. Random calculation in IBSimVis, is done using the `boost::random` library and Mersenne twister [71] implementation with the current time in nanoseconds as seed. The `LayoutAlgorithm` superclass defines a purely virtual function called `layout`, which all classes that inherit from `LayoutAlgorithm` need to implement. The purpose of the `layout` function is to initiate the implementation of a given layout algorithm. However, due to time constraints we were not able to finish implementing the multilevel part of the algorithm. Hence, we revert to a single-level version, similar to that of Fruchterman and Reingold, mentioned in Section 2.3.5.1. The main difference is that the graph coarsening phase is omitted from our implementation.

Pseudocode 5.2 Pseudocode of the repulsive, attractive and cooling functions in our single-level implementation of Walshaw's multilevel force-directed placement algorithm.

```
1: function FR(w, x, k)
2:     return −Cwk²/x;
3: end function
4:
5: function FA(x, k)
6:     return x²/k;
7: end function
8:
9: function COOL(t)
10:    λ = 0.99;
11:    return λt;                                    ▷
12: end function
```

As mentioned above, we only provide a part-implementation of a the multi-level force-directed placement by Walshaw, based on the pseudocode in Figure 1 of [29]. In the article the author mentions changes that can be done, to implement a single-level version with the layout looking similar to the multilevel one [29], namely:

- Set node weight to 1
- Increase λ during the cooling phase

Our implementation is shown as pseudocode in Pseudocode 5.3 and our changes are marked in red. We use a node weight of 1, as Walshaw indicates when one is to implement a single-level version [29], this is shown on line 12 in Pseudocode 5.3. We found that the layouts generated by the single-level implementation so far, did not create satisfactory layouts. But according to Walshaw, a single-level scheme would produce the same result as a multilevel sheme, by adjusting the cooling schedule [29]. This means increasing the λ in the cooling function, hence also increasing the amount of iterations. Walshaw defines the amount of iterations, i as $\lambda^i < tolerance$ [29]. Where both λ and *tolerance* are constants, set to 0.9 and 0.01, respectively. Since originally,

Pseudocode 5.3 Pseudocode of our single-level implementation of Walshaw's multilevel force-directed placement algorithm. Based on Figure 1 in [29]

```
1: function LAYOUT
2:     Posn = Topology→graph;
3:     k = initialspringlength;
4:     t = k;
5:     tol = 0.01;
6:     while converged ≠ 1 do
7:         converged = 1;
8:         for all v ∈ V do
9:             Θ = 0;
10:            for all u ∈ V, u ≠ v do
11:                Δ = Posn[u] − Posn[v];
12:                Θ = Θ + (Δ/||Δ||) · FR(||Δ||, 1, k);
13:            end for
14:            for all e ∈ neighbours do
15:                Δ = Posn[u] − Posn[v];
16:                Θ = Θ + (Δ/||Δ||) · FA(||Δ||, k);
17:            end for
18:            newPos = Posn[v] + (Θ/||Θ||) · min(t, ||Θ||);
19:            oldPos = Posn[v];
20:            Posn[v] = newPos;
21:            Δ = newPos − oldPos;
22:            if ||Δ|| > k · tol then
23:                converged = 0;
24:            end if
25:        end for
26:        t = COOL(t);
27:    end while
28: end function
```

the multilevel algorithm used the same cooling scheme as the Fruchterman-Reingold algorithm, and the constants in the cooling schedule was found by "extensive experimentation" [29], Section 2.3.5. We simply adjusted the λ in the cooling function from 0.9 to 0.99, as shown in Pseudocode 5.2, line 10. We found out that with λ of 0.99 and *tolerance* of 0.01, we could get good looking layouts. The amount of iterations would then be:

$$0.99^i < 0.01$$

Leading to:

$$i = \frac{\log(0.01)}{\log(0.99)} = 458.210577$$

Note that using this cooling schedule, it might be subject to floating point rounding errors. So we do not necessarily get 458 iterations in our implementation. This also means we know an upper value of the number of iterations. Enabling us to easily construct a progress bar, to show the user how long time is left of the layout calculations, which we mentioned in the design.

The only difference from our implementation to that of Walshaws pseudocode, barring the single-level changes, is that we do not copy the graph before running the repulsive and attractive calculations. We did not do this since we use pointers in the nodes of the graph to directly manipulate the member variable in the `Node` class that contains positional data. Instead, we store a variable containing the old position in a variable called `oldPosition`, in the repositioning phase, as shown in line 19 of Pseudocode 5.3. Then, we reposition the node and calculate the delta. We do this instead of copying the entire graph, which we did in the first prototype of IBSimVis, since copying a BGL graph has a time complexity of $O(|V| + |E|)$ [68]. One more thing to note about our implementation of the multilevel force-directed placement algorithm of Walshaw, is that we use the `Ogre::Vector3` library abstraction for linear algebra, instead of a general one. This inherently ties our Layout component implementation closely to the OGRE library.

As mentioned in our design, we also want to use the iteration calculation

above, to return the number of iterations to supply to a progress bar. For each iteration, the progress bar is updated and we also know the max number of iterations. We use the `boost::progress_display` to implement the progress bar, alternatively it could be shown in the GUI.

We reserve finishing the multilevel algorithm implementation as future work, in addition to adding other layout algorithms to the Layout component.

5.4 Visualisation Component implementation

In this section, we describe our implementation details in regard to the Visualisation Component (VC). The design for the VC was discussed in Section 4.4. We also discuss what choices we had to take when implementing the Graphics Engine component, such as selecting a graphics engine library.

5.4.1 Graphics Engine

As mentioned early in this chapter, there are many choices of graphics engines available to developers [63], using the C++ language. Some are light wrappers around OpenGL, being able to draw graphic primitives in an highly efficient manner. Some are *rendering engines* with higher-level abstractions wrapping around OpenGL. And other graphic engines are *game engines* complete with implementations for physics-, input- and sound- support.

In the first IBSimVis prototype, we tried out two different graphics engines based on popular choices at [63]. These were OGRE [72] and the Visualization Toolkit (VTK) [73]. OGRE is an open-source cross-platform scene-based 3D rendering engine written in C++, used for writing applications within games, simulations, visualisations or other business applications [72]. VTK is an open-source cross-platform system, written in C++ for 3D computer graphics, image processing and visualization, with a rich feature-set for scientific visualisation [73].

Since one of the requirements of IBSimVis, mentioned in Section 4.1.2, required that it could display a network topology and able the user to interact with it, we had a set of requirements for the graphics engine itself. Namely:

- Able to render 2D and 3D
- Provide high-level abstractions to graphics primitives
- Provide abstractions for interaction in 3D
- Be easy to integrate with the GUI
- Be cross-platform
- Have good documentation
- Have an active user community

In the end, the choice fell on OGRE as a graphics engine in IBSimVis. The reason for this is that it fulfills all the requirements mentioned above. It comes with renderers for both OpenGL and Direct3D, allowing for cross-platform compatibility, in addition to being easy to implement. The main advantage for a scene based design is that one can easily organise and apply transformations to the nodes. Another factor is that OGRE has a visible and active user-community, contrary to VTK which relies on subscription mailing lists [73]. As we see in a bit, searching the OGRE forums helped us solve a problem when integrating with the GUI library, wxWidgets. We also saw that VTK was more tuned for applications within scientific visualisation, rather than information visualisation.

In the next few sections, we first describe the OGRE library, as some knowledge about it is required in order to understand our VC implementation. Then we move on to describing our overall VC implementation and how we implement the encoding mechanisms to the enitities by rendering nodes and links in our static network topology. Afterwards, we move on to the challenge of interacting in a 3D environment.

5.4.1.1 The OGRE library

As mentioned above, OGRE is a rendering engine, used for 3D applications. In this section, we briefly describe central OGRE library concepts, and classes

that have been used when implementing the VC of IBSimVis. In addition to the typical program flow when using OGRE.

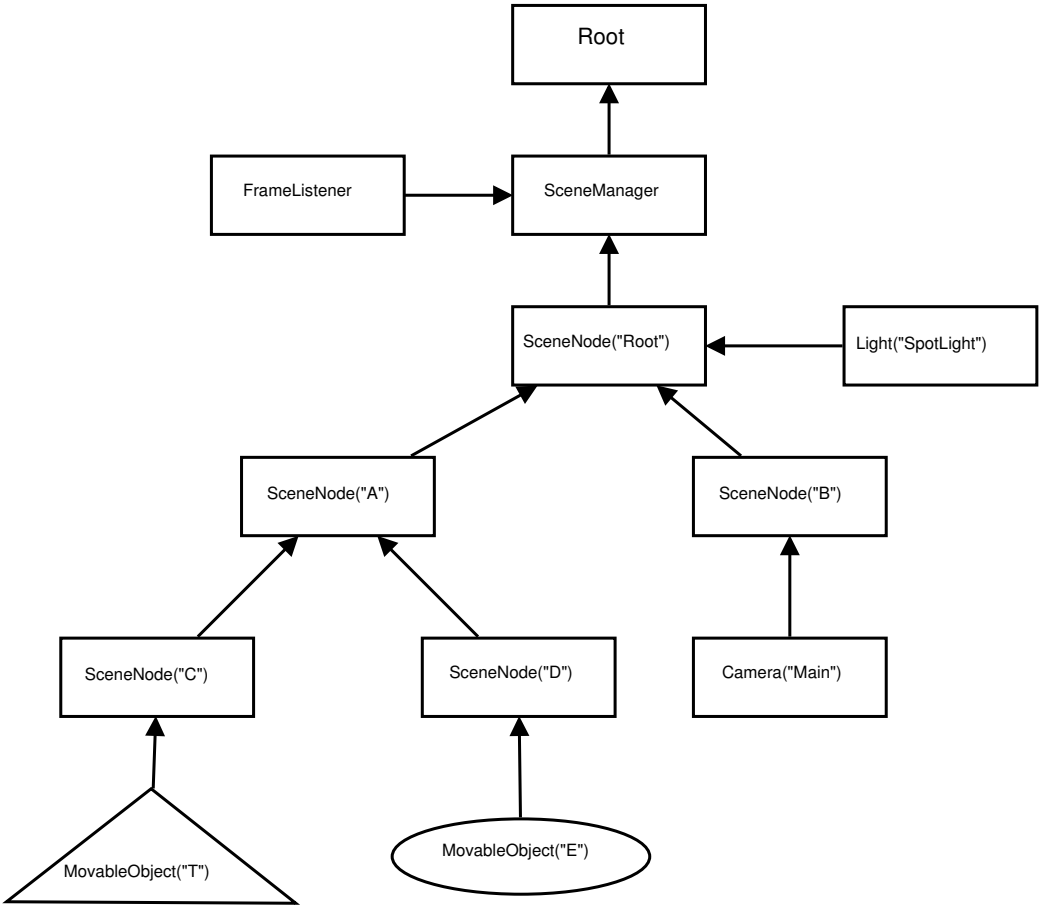


Figure 5.2: Example of a scene graph in OGRE. Showing relationship between the `SceneManager`, `SceneNodes` and world objects such as `MovableObjects`.

As seen in Figure 5.2, the `Root` object is the most central class of OGRE. It acts as a hub controlling all fundamentals in an OGRE system. The `Root` class manages the `SceneManager` class, whose main task is to control the internal scene graph, comprised of scene nodes using `SceneNode` objects. As illustrated in Figure 5.2 one can attach world objects such as glyphs, cameras and lights to a scene node. Transformations (translation, rotation, scaling) applied to a scene node also propagate down to its children. For example, rotating the "A" scene node in the figure, would rotate both the

world objects “T” and “E” and translating (moving) the scene node “B” would move the “Main” camera. Furthermore, the `SceneManager` indexes both scene nodes and world objects using an unique string, requiring the programmer to keep track of what a given scene node controls or what a given world object represents.

A `ManualObject` in OGRE is a world object and provides a simple interface to creating OpenGL-style [48] custom geometry [74]. This means that one has to call `position`, `colour` and `normal` for each vertex. We decided to render entities in IBSimVis using these basic world objects, since they are very fast, with a low memory footprint. At the cost of few features and higher-level abstractions.

A `FrameListener` is the input handler in OGRE, designed to consume input such as mouse events and keyboard strokes, in addition to special cases between each time a frame is rendered. In an application, one typically creates a class that inherits from the `FrameListener` class in OGRE. OGRE is typically initialised in several stages, usually in the following order:

1. Initialise the `Root`
2. Get the available render system of the platform and initialise one (OpenGL or Direct3D).
3. Define resource locations for assets, such as textures, meshes and sounds.
4. Create the `SceneManager`
5. Create camera and lights
6. Create viewport
7. Create scene
8. Create `Framelistener`
9. Initiate render-loop

After initialisation, OGRE renders one frame at a time, and listens for user input between frames. This repeating sequence is referred to as the

render loop [75]. Also note that during the stage where the viewport (the viewing area of the camera) is created, the background colour is also set. For further information about the OGRE framework, we refer to the website [72], wiki [75], API documentation [74] or book [76].

5.4.1.2 Applying encoding mechanisms to entities

As mentioned at the start of this chapter, we did not have time to implement simulation data visualisation features, so we do not take into account message entities. And for reasons mentioned in Section 4.4.4 in the design, port entities are not converted to glyphs. Thus, the only two entities we are interested in applying encoding mechanisms to, is `Node` and `Link`. To encode entities, we use a class containing static methods using a factory-like design pattern to encode shapes and colours to a world object. These glyphs form the basic structures in our visualised network topology. Moving on, we cover how we implemented the encoding mechanisms and how we applied them to the `Node` and `Link` entities managed by the DMC.

As mentioned in the design, we want to encode `Node` entities as coloured shapes. Switches are encoded as blue cubes and CAs are encoded as red spheres. The way the encoding mechanism is applied is using OpenGL-like syntax [48] by first denoting each vertex position and colour. Since we use no lighting or shading on the objects, there is no need to apply normals, although this can easily be applied in the future. The advantage of this approach, is that both shape and colour can be applied during the same procedure. The disadvantage of this approach is that, to change colour on a node, one has to update the `ManualObject` by calling position (with the same position) and colour (with the new colour) again.

Link entities are encoded as coloured straight-lines, using `ManualObject` and a line list from a position to a position. This way, OGRE automatically draws a line between these two points, which is always one pixel wide, regardless of viewing distance [74]. Listing A.6 in Appendix A shows how encoding `Link` entities were implemented.

5.4.1.3 Visualising the static network topology

Our implementation of the OGRE initialisation is done in the function `initOgre` in the class `wxMainFrameImpl`, which is the main entry point in IBSimVis. We use the OpenGL renderer in OGRE, since we program on a GNU/Linux system. OpenGL can also be used on other platforms, with some adaptions to our implementation. The implementation of the input handler is done in a class called `wxOgreFrameListener`. Our implementation of the VC first starts with the initialisation of OGRE following the same pattern of stages as mentioned earlier. The background colour is set to black in the viewport stage, as per the design. And we use a right-handed coordinate system, looking down the negative z-axis from the camera. When that is done, we start the render-loop, using the wxWidget input system and event handler to take care of keyboard/mouse input events. This is passed to the OGRE render window and pushed further to our custom OGRE input handler.

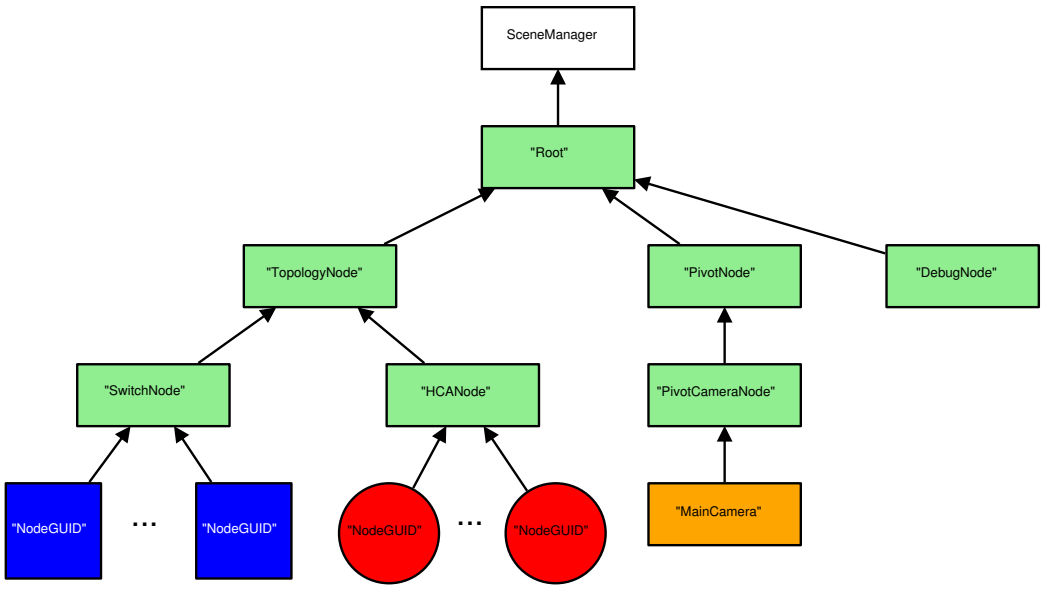


Figure 5.3: The scene graph structure in IBSimVis, showing the scene graph structure from the SceneManager.

We do not go into detail of all the library calls required to initialise OGRE. But instead describe how we render the scene, the display area where the network topology is visualised and show how it is organised. This is done in

`wxMainFrameImpl`. Here, we create the scene nodes that comprise the scene and then proceed to render nodes first, with links afterwards. `Node` entities are indexed in the `SceneManager` with the key being a string representation of their GUID. While `Link` entities are indexed using a key with the following format:
"port1GUID:port1PortNumber,port2GUID:port2PortNumber"
In our first prototype, we used a simple key in the format of "NodeGUID1,NodeGUID2". This was insufficient, since we in some topologies, we encountered multiple links between nodes. The keys for the `Link` entity mentioned above, only shows one direction of a link. To retrieve the other direction, we simply swap what is between the comma. Note that OGRE does not know the difference between a `Node` or `Link` entity in its `SceneManager`, it was our task to create keys that can identify them. When all nodes and links are placed in the scene, we calculate the bounding box of all the nodes and attempt to center the topology. The reason why we do this, is that during layout, nodes might have been placed outside the viewing volume. It would be inconvenient if the network topology for example was rendered behind the camera. Figure 5.3 illustrates how the scene in the VC is organised. Here we can see the green SceneNodes, orange cameras and the entities we applied encoding mechanisms to, mentioned earlier. The "TopologyNode" holds the "SwitchNode" and "HCANode", keeping track of switch and HCA representations, respectively. The "PivotNode" is in the center of the scene. Attached to that is the "PivotCameraNode", which the camera "MainCamera" is attached to. What is not shown in the illustration, is the links that are attached to either the "SwitchNode" or "HCANode", depending on which node type they belong to.

5.4.1.4 3D interaction

In Section 4.4.1 and Section 4.4.6, we learned that navigating and interacting with 3D can be problematic. Here we address how we implemented rotation, geometric zoom and pan in IBSimVis. In addition to how the Entity Selection Query works and how it is implemented. All these functions were implemented in our input handler class. This class receives mouse and

keyboard events from the wxWidgets event handler system, via the widget `wxOgreRenderWindow`, described later.

Geometric zoom

In the design we stated that we wanted to use geometric zoom. As we can see in Listing 5.4, we simply use the wheel-delta of the mousewheel and move the camera along the z-axis only. When the mousewheel is scrolled in a forward motion, the wheel delta is positive. If it is scrolled backwards, the wheel delta is negative. According to the mouse event documentation in wxWidgets, the scrolling delta is always 120 [77]. We increase this by a factor of 1.5 to enable abit faster scrolling. Since we look down the negative z-axis, a negative wheel delta would make the camera move along a positive z-vector. The opposite is true when scrolling forwards.

Listing 5.4: The zoomCamera function in the `wxOgreFrameListener`.

```
void wxOgreFrameListener::zoomCamera(int delta)
{
    mCamera->moveRelative(Ogre::Vector3(0.0f, 0.0f, -(delta /
        120.0f) * 1.5f));
}
```

Pan

Panning is implemented in the `panCamera` function in our `wxOgreFrameListener` class. The user can pan when the right mouse button is pressed down. It takes the relative x and y mouse positions as parameters. These are calculated by aquiring the difference between the current mouse position and the old mouse position. To pan, we move the camera along the x and y axis.

Listing 5.5: The panCamera function in the `wxOgreFrameListener`.

```
void wxOgreFrameListener::panCamera(int relX, int relY)
{
    mCamera->moveRelative(Ogre::Vector3(-relX, relY, 0.0f));
}
```

Rotation

Rotation is done when the user holds down left mousebutton in the visualisation window. There are several approaches to rotating a 3D model. One involves rotating the camera around the model and the other rotating the model with the camera being static. We choose an approach similar to the former as a matter of preference. Here, we rotate the "PivotNode", that the camera looks at, thus allowing the model to be still while the camera is able to rotate at a fixed distance. As shown in Listing 5.6 in Appendix A, when the differences in x and y changes, we pitch and yaw the `SceneNode` "PivotNode" mentioned earlier, so that `SceneNode` is pitched and yawed. Note that we use a 2D technique to rotate a 3D model, an alternative implementation to rotating a 3D model, could be to implement a virtual trackball instead [78].

Listing 5.6: The rotateCamera function in the `wxOgreFrameListener`.

```
void wxOgreFrameListener::rotateCamera(int relX, int relY)
{
    Ogre::Degree rotX = Ogre::Degree(-relX * 0.30);
    Ogre::Degree rotY = Ogre::Degree(-relY * 0.30);

    mSceneMgr->getSceneNode("PivotNode")->pitch(rotY);
    mSceneMgr->getSceneNode("PivotNode")->yaw(rotX);
}
```

Entity Selection Query

The Entity Selection Query is used when we want to select objects in 3D space. Selection is done in IBSimVis, with the user left-clicking an entity, whereas the function `selectedObject` is called in `wxFrameListener`. The main challenge this function deals with, is how to select 3D objects using a 2D display and 2D input controller (typically a mouse). In addition, 3D objects might occlude each other. Simply selecting a 3D object by its x and y position is not enough, since some 3D objects are in the foreground and others in the background. One has to take into account the axis that the camera is aligned with.

Our implementation uses the OGRE `Ray` and `SceneQuery` class functionality. Briefly explained, a ray is shot out from where the mouse is clicked, where it can intersect a series of `MovableObjects`. Then we query the `Ray` using a `SceneQuery` by sorting the `MovableObjects` the Ray intersected, by depth and filtering them by type. The query mask used as a parameter, is a mask that is applied when creating the entity representations, to differentiate between glyphs. For example, a node has the query mask NODE_FLAG and link, LINK_FLAG. Our implementation of the Entity Selection Query is shown in Appendix A, Listing A.5.

5.5 Graphical User Interface

In Section 5.2, we mentioned that we changed our design to conform to the design of our chosen GUI library. In this section, we discuss why we chose that specific GUI library, wxWidgets [79] by comparing it to two other viable alternatives, Crazy Eddies GUI (CEGUI) [80] and Qt (pronounced "cute") [81]. Then we describe the wxWidgets library and central features used in IBSimVis. Afterwards, we introduce the implementation of our three main views, namely the normal, NumPaths and Routing views.

We briefly introduce the GUI libraries we considered for IBSimVis. For more extensive information, we refer to their websites [79–81]. wxWidgets is an open-source cross-platform widget toolkit for creating GUIs, implemented in C++ with bindings to many other programming languages [79]. It offers a native look and feel, since it uses the native API of the platform it runs on, instead of emulating the GUI [79]. CEGUI is an open-source cross-platform widget API for games, implemented in C++. It is designed to offer a GUI to graphics engines, where that feature is missing, and it is typically rendered inside a 3D environment, such as OpenGL, OGRE or Direct3D. Qt is an open-source cross-platform widget toolkit, implemented in C++ with bindings to many other programming languages [81]. Recent versions of Qt, features native platform API calls for drawing the GUI.

When deciding between the GUI libraries mentioned above, we had two main critera. The first, that it had to to work well with OGRE, using either

of two methods, drawing the GUI inside OGRE or embedding OGRE in a widget. The second, that it had to have core GUI features present in desktop applications. For example, filemenu, statusbar, toolbar, treelist, checkboxes, radio boxes and input fields. In the first IBSimVis prototype, we used CEGUI, as it was the GUI library recommended by the OGRE community. Mainly because it has a dedicated OGRE-renderer. However, it is aimed at game development and fell short when it came to core features present in desktop applications, such as the features required above. We also found it hard to use with drawing fonts correctly. So we had to look for a new GUI library, having to choose between wxWidgets or Qt. Both can embed OGRE in a widget, both natively draw GUI depending on the platform, both have a What You See Is What You Get (WYSIWYG) GUI editor for rapid widget prototyping, both are well documented, both have a vibrant and active user community and both have less restrictive licenses (Both Qt and wxWidgets has the Lesser GNU Public License (LGPL) based licenses.). In the end, we could not find any compelling arguments for choosing either, so we settled on wxWidgets because of personal preference, since we had some prior programming experience with that toolkit, cutting down on time needed to learn Qt. Next, we describe the wxWidget toolkit.

5.5.1 The wxWidgets toolkit

As mentioned above, wxWidgets is a widget toolkit. Containing all the widgets that a typical desktop environment has. In addition to the features mentioned above, it also has a flexible event handler for GUI events, enabling simple and easy communication with the underlying application data model.

IBSimVis does not have a `main` function as an entry point. Instead, wxWidgets wraps around the `main` function, making it heavily integrated to the rest of the application. A wxWidget application is a hierarchy of windows and widgets, base visual objects on the screen [77]. After wxWidgets initiation, it runs the main thread and initiates the event handling [77], described below.

The wxWidgets event system uses event tables, instead of programmatic

callbacks to map events to window member functions [77]. Each interactable window in wxWidgets has its own event handler. An event is declared in the event table with a window ID and a member function belonging to a window [77]. A window ID is an int, typically declared in an enum and might identify a button in a panel. Listing 5.7 shows an event table used in wxWidgets, with the event, `EVT_BUTTON`, indicating someone clicked a button widget with `EXIT_BUTTON` window ID, calling the exit member function in the MyFrame window. It is the programmer that has to keep control over which window IDs that is assigned to which window. When a user clicks any visual element, the wxWidgets window system fires an event off to the widgets event handler, scanning the event table to call the appropriate member function. If no event table entry is found, the event returns and nothing happens. The exact method of searching event tables is detailed in the "Event handling overview" of [77].

Listing 5.7: An event table in wxWidgets, showing how it binds events to functions.

```
BEGIN_EVENT_TABLE(ExampleFrame, wxFrame)
  EVT_BUTTON(START_BUTTON,  ExampleFrame::start)
  EVT_BUTTON(EXIT_BUTTON,   ExampleFrame::exit)
END_EVENT_TABLE()
```

Similar to for example Qt, wxWidgets has its own string implementation, instead of the C++ `std::string`, in a class called `wxString`. It provides 90% of the `std::string` functionality, in addition to giving full unicode support [77]. The disadvantage is that the programmer has to convert the `wxString` to a `std::string` when it is used, either in the application or in third-party libraries. For more information about wxWidgets and their widgets, we refer to the website [79], documentation [77] or the book [82].

5.5.2 Run-time configuration

For IBSimVis to process input files, we need a way to input them to it. This is implemented as a standard command-line option parser, using the

wxWidgets command line parser. In the first iteration of IBSimVis, we used `boost::program_options`, but decided to scrap this implementation as wxWidgets had built-ins for command-line option parsing and configuration file parsing. Using the option -t followed by the filename of the topology data file, one inputs this to IBSimVis. And the option -f followed by the routing data file, inputs this to IBSimVis. Presently, we have not defined a configuration file, we leave this as further work.

5.5.3 Implementation

Earlier we mentioned that we had to do a major change in our original design to conform to the application structure that wxWidgets enforces. Namely that wxWidgets is the entity that both initialises and acts as the application control flow. However, the same behaviour that wxWidgets exhibits, is also present when using Qt [83]. An advantage to this approach is that the GUI is able to directly communicate with the DMC and the VC, instead of going through the VC as originally proposed in Section 4.2. The disadvantage to this approach is that we always have a tight coupling to wxWidgets and swapping the GUI to use for example Qt in the future, might be a tedious task. Just like Qt, wxWidgets has a WYSIWYG GUI design editor, to make it easier to design GUI panels. This is called wxFormBuilder [84]. Instead of tediously designing the layout of GUI panels programmatically, we used wxFormBuilder to generate widget layout code, when creating the GUI in IBSimVis.

In our design, we wanted to create an event loop for input listening and event handling. Currently, as mentioned in Section 5.5.1, wxWidgets takes care of this, using framework specific input handling. Sometimes we mention the classes, functions and terms used when implementing the other components. We refer to the sections describing those components when we mention such terms here. Next, we will briefly explain the application flow of IBSimVis.

1. Read user command-line input

2. Initiate `wxMainFrameImpl` (the application window frame)

3. Initiate the `Topology` class

4. Parse topology and routing data, using `TopologyParser`

5. Run Layout component, using a `LayoutAlgorithm` (`MLFDP`)

6. Initialise OGRE

7. Calculate path distribution using Topology component

8. Launch the application window

When program options have been input to IBSimVis, wxWidgets initialises in the `wxMainApp` class, being the entry point. Then we parse the command-line using the built-in command-line parser. The command-line options are put in a simple struct, called `AppOptions` and passed to the main GUI window, `wxMainFrameImpl`. The main window initialises the `Topology` object and proceeds to initiate `TopologyParser`, with the topology data file, routing data file and `Topology` object as parameters. It then tells the Parser component to parse the topology data. When that is done, it creates the `MLFDP` object and spawns two threads: one to parse the routing data file and one thread to layout the network topology. We can do this in parallel without suffering from race conditions, since parsing routing data does not touch the positional attribute in the `Node`. After both the threads are done, we route all CA to all CA.

After the setup phase, we proceed to initialise OGRE, as described in Section 5.4.1.2. The communication between the GUI and the VC is mainly done via the `wxOgreRenderWindow` class, which is contained inside a wxWidget widget as seen in Figure 5.1. This is the class that offers the user a 3D topology visualisation. After OGRE initialisation, we create all the views and enables the user to see the visualisation and the ability to choose between views.

5.5.4 Implementing views

The normal view is simply the default OGRE visualisation and offers no extra options. We have already described this implementation. The normal view is implemented in the `wxNormalViewPanel` class.

5.5.4.1 Routing view

The Routing view is implemented in the `wxRoutingViewPanel` class. It offers two input text fields, where one is used for the source GUID address and the other is used for the destination GUID address. Here, the user can paste in which two CAs he wants to route between. There are also checks in place, to make sure the user can not route to or from switches. A warning dialog warns about operations that are not allowed. Such as routing from a CA to a switch. A button labelled "Route" is implemented. When the user clicks on this, the GUI attempts to call the Topology component to retrieve the `RoutingHistory`, an object that manages a collection of `RoutingHop`. When the route has been obtained, the Routing view signals to the VC that the route should be highlighted in green, by changing the colour of the link. In addition, only neighbouring nodes and links to each hop is shown, the rest are hidden.

5.5.4.2 NumPaths view

The NumPaths view is implemented in the `wxNumPathsViewPanel` class. It offers a colour pallette as mentioned in the design, allowing to the user to choose between interval colours. The intervals are implemented per the design, dividing for example the $link_{max}$ number of paths into 5 steps, each step shown as a different colour. In addition, it offers two Dynamic Query sliders, to filter the number of paths of links and nodes respectively. The sliders are independent of each other, so one can decide to filter links and then filter nodes. The sliders are initiated by the statistics gathered in the `Topology` class, having ranges between $link_{min} - link_{max}$ and $node_{min} - node_{max}$ for each of the sliders.

The interactive element of the NumPaths view is the Dynamic Querying. The way it is implemented is by using a `std::map` with the number of paths as a key and a `std::vector` of the nodes that have the same number of paths as the key. This way, we can exploit the ordering that `std::map` automatically does, since internally it is a binary search tree, having a *natural ordering* of the keys, which are numbers. For example, if we drag the node filter slider we listen to the slider drag event. When the slider drag event has been caught by the event handler in `wxNumPathsViewPanel`, we hide the entire network topology and filter out all nodes that are greater than or equal than the value the slider is currently at. So for each slider drag event, we do an $O(n)$ iteration through the `std::map`, hiding or showing the nodes.

5.5.5 Implementation issues

During the implementation of the GUI, we encountered a few issues around embedding OGRE in a wxWidgets widget.

The `wxOgreRenderWindow` class we used are based on the class with the same name in another open-source project, OgreMeshy [85] which again is derived from `wxOgreControl` in the open-source wxOgre project [86]. When we first implemented this class when working on the second iteration of IB-SimVis, we noticed a massive performance drop in Frames Per Second (FPS) compared to our first prototype, where we only used OGRE and CEGUI. We also noticed that our mouse and keyboard input would not be registered. Searching the OGRE forums, we found a solution to our problem [87]. The reason for the FPS drop was two bugs in the class we had re-used. The implementation previously used in `wxOgreRenderWindow`, was timers to refresh the OGRE 3D scene. Each time the timer timed out, the frame would updated. The reason for the FPS drop was that the code handling frame updates in the `wxOgreRenderWindow`, called `renderOneFrame` and `update` in the same timeout callback. The two functions operate in a similar manner, enforcing frame updates, thus forcing two frame updates in the same callback. The timer based approach was not good, since wxWidgets would then use all idle processing time to update the renderwindow, thus blocking the input devices.

Table 5.3: Used libraries, their licenses and whether they are GPL compatible or not

Library	License	GPL compatible
Boost C++ libraries	Boost Software License	Yes
OGRE	MIT License	Yes
wxWidgets	wxWindows License	Yes
wxOgre/OgreMeshy	GPLv2	Yes

Having found out the reason why this happened, we created a callback for the `wxOgreRenderWindow`, as suggested in the forum thread [87], registering it with an idle event in wxWidgets. So every time every time the application was idle, it would enter the callback and not obstruct user interaction. To fix the frame update problem we simply removed the call to `renderOneFrame`.

5.6 Licenses

In this section, we will briefly cover the licenses used in the implementation of IBSimVis. Since we have used both dynamically linked libraries, headers and classes from open-source projects. The libraries used and their licenses are summarised in Table 5.3. As we can see from the table, each library uses a different license. We will not describe each license, since they are not the focus of this thesis. But instead, we point out that we have used the GNU General Public License (GPL) version 2 code in our implementation. The GPLv2 license states that derivative works must also be under the GPL. Thus, all source code in IBSimVis needs to have the GPLv2 license. The other libraries licenses also do work fine with GPL. In addition we had to notify about the other library licenses in our distribution, as per the text in their licenses.

5.7 Summary

In this chapter, we presented a detailed description of all the components and inner workings of IBSimVis. We saw how the design differs from the implementation, with the GUI component acting as a supercomponent controlling the other components. In the Parser component we used simple regular expression matching with `boost::regex`. In the Topology component, we decided to use the BGL because it is a fast and highly configurable graph library. The BGL is only used by the Layout component for graph drawing. In the VC we used OGRE as the Graphics Engine. For the GUI component, we chose wxWidgets. In the next chapter, we move on to see how IBSimVis looks like and how it can aid a researcher in solving problems related to issues within network topology, routing and path distribution.

Chapter 6

Evaluation

In this chapter, we present the result of our thesis: IBSimVis. We show what it looks like using different network topologies and attempt to evaluate its usability. We start off with the evaluation environment in Section 6.1, introducing what methods, tools, topologies and testbed we have used to show get our results. In Section 6.2, we continue by taking a first look at IBSimVis and evaluate its usability. At the end, in Section 6.3, we provide a summary of our results and evaluation.

6.1 Evaluation environment

6.1.1 Method and tools

We have done a qualitative evaluation of IBSimVis based on its usability in visualising and analysing network topologies and routing data. We have also done a quantitative evaluation of how the application scales in time compared to the number of nodes present in the topology. Note that the preferred method of evaluation was to do a user study on individuals, both with and without knowledge about IBSimVis's application area and collect information using interviews or questionnaires. However, we could not commit to those methods due to time constraints.

In addition to IBSimVis itself, we have used is the UNIX command `time` to measure various aspects of our application. We used `gnome-panel-screenshot`

Table 6.1: The name and properties of the sample topologies used to evaluate IBSimVis.

Topology name	$\|V\|$	CA	Switches	$\|E\|$	Regular topology?
4-ary 2-tree	24	16	8	32	Yes
4-ary 3-tree	112	64	48	192	Yes
4-ary 4-tree	512	256	256	1024	Yes
TITAN	885	581	304	2348	No

to take screenshots and `gnuplot` to plot charts.

6.1.2 Sample topologies

We have decided to use a selection of both regular and irregular topologies, to see how well IBSimVis can handle different classes of network topologies. The regular topologies are all k-ary n-trees, n ranging from 2 to 4, of which two were previously shown in Section 2.1.1.3. The main motivation for evaluating IBSimVis using fat-trees is that they are the preferred topology for InfiniBand networks [15], as mentioned earlier. The irregular topology we have chosen, is the one used by the University of Oslo's Research Computing Services (RCS) group, called the TITAN computing cluster [88]. It is a Linux-based computing cluster with 581 compute nodes able to do about 40 teraFLOPS currently, using InfiniBand as interconnect. The topologies and their network topology properties, are summarised in Table 6.1.

6.1.3 Testbed

To evaluate IBSimVis, we used a PC workstation, with two single-core Intel®64-bit Xeon™3.00GHz CPUs, using symmetric multiprocessing (SMP), allowing for more efficient use of multithreaded applications. The two CPUs share the same 4GiB main memory. The graphics card is an nVidia Corporation NV41GL Quadro FX 1400 with 128 MiB memory. We realise that this hardware is currently abit dated, being from 2005. However, this shows that IBSimVis can run on older hardware. Newer and better hardware most likely

is going to increase IBSimVis performance. The Operating System (OS) used is Ubuntu 10.04 *Lucid Lynx*. Currently IBSimVis has only been tested on a Linux distribution, however all software dependencies are cross-platform capable and could be easily extended to other platforms in the future. All screenshots are taken in 1280x1024, IBSimVis's default resolution. However, the IBSimVis application window can be resized in case of higher resolution monitors.

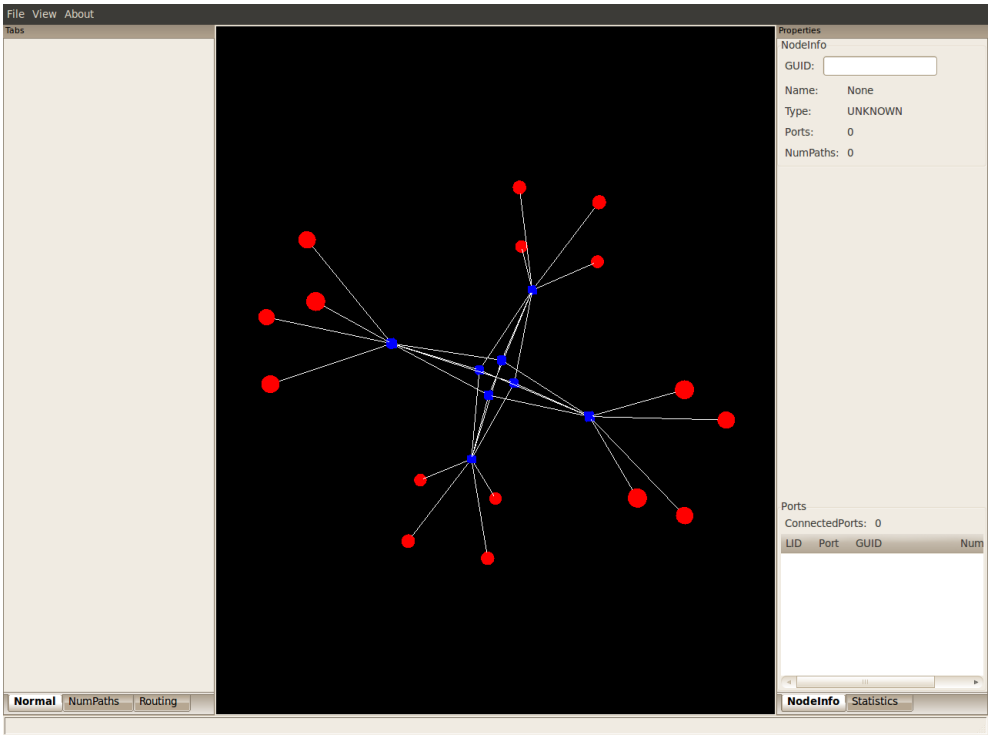
6.2 Uses and results

In this section, we show the results of our design and implementation. Namely, a visualisation tool prototype for OpenSM-defined topologies and routing data. We start off by showing a first impression of the topologies mentioned in Section 6.1.2, as visualised by IBSimVis. Then move on to evaluate various aspects of it.

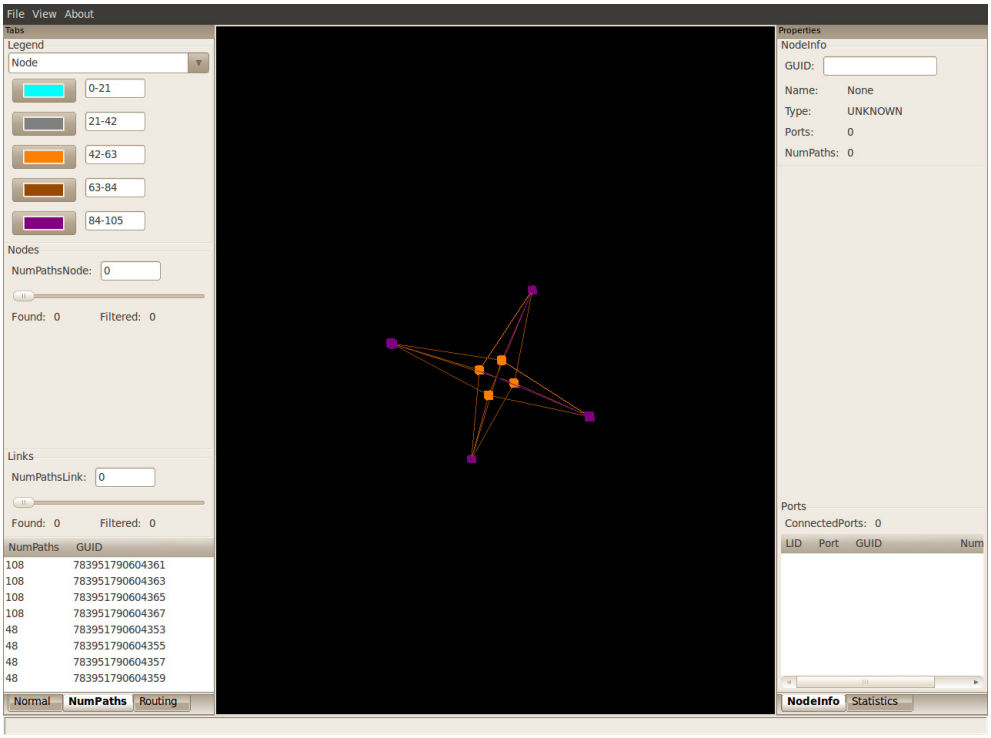
6.2.1 First impressions

Here we present a first look at IBSimVis in its current form, showing both the Normal and NumPaths view for each of the network topologies mentioned above. This is shown in Figure 6.1 (4-ary 2-tree), Figure 6.2 (4-ary 3-tree), Figure 6.3 (4-ary 4-tree) and Figure 6.4 (TITAN). The images were taken after rotating, zooming and panning to fit the whole network topology in the VC widget. Note that the figures in this chapter might not print well on paper, with the printer resolution not being able to show the structural relations in the network topology or colours properly. For this reason, we have provided an archive of all the figures, available at our website[1].

[1] `http://heim.ifi.uio.no/~joakibj/master/figures.zip`

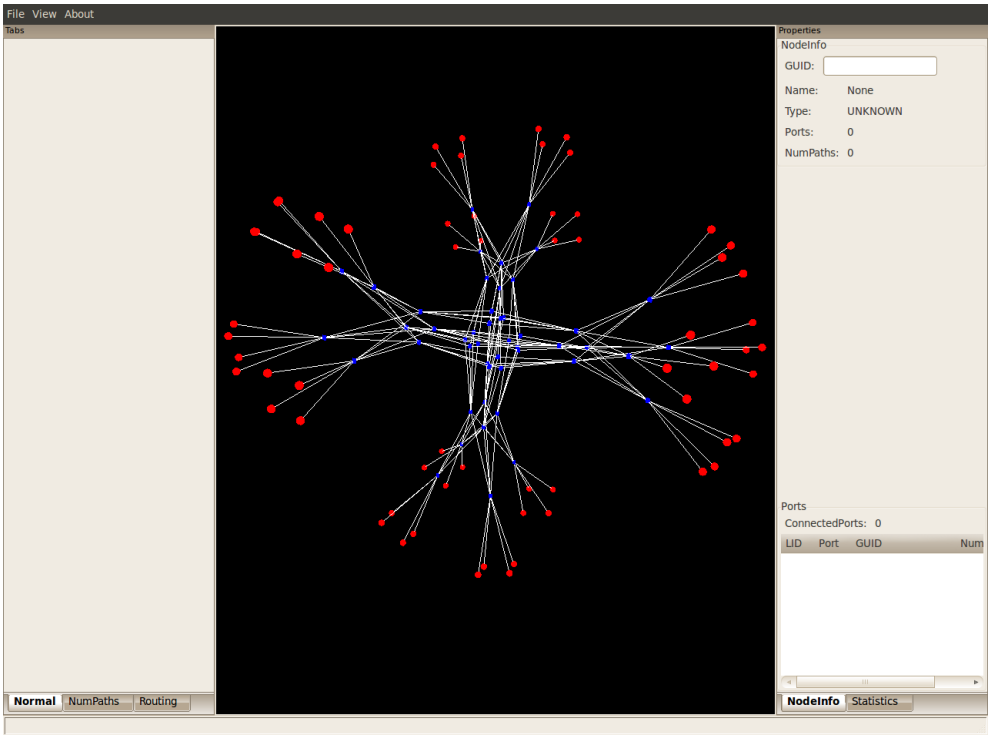


(a) 4-ary 2-tree visualised by IBSimVis, using Normal view.

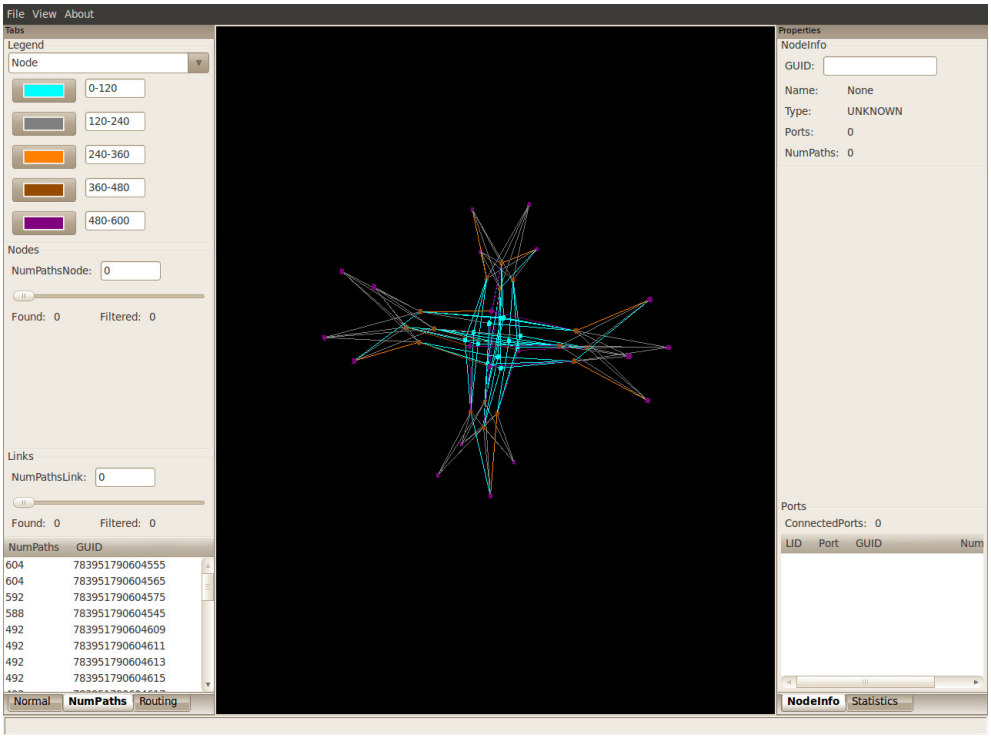


(b) 4-ary 2-tree visualised by IBSimVis, using NumPaths view.

Figure 6.1: 4-ary 2-tree visualised by IBSimVis.

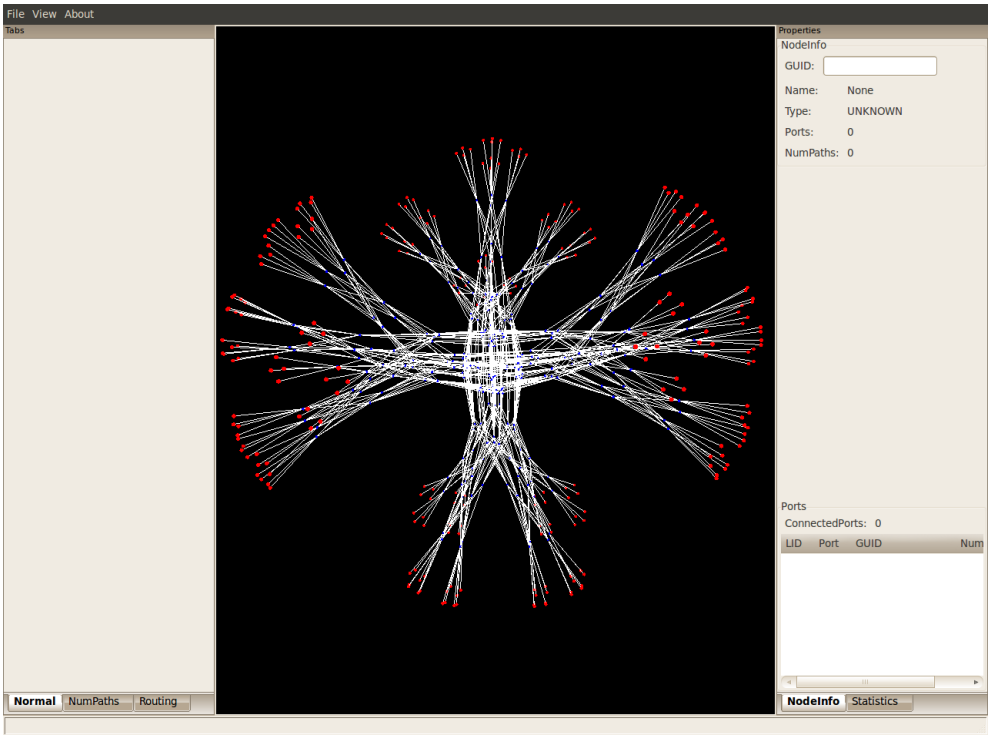


(a) 4-ary 3-tree visualised by IBSimVis, using Normal view.

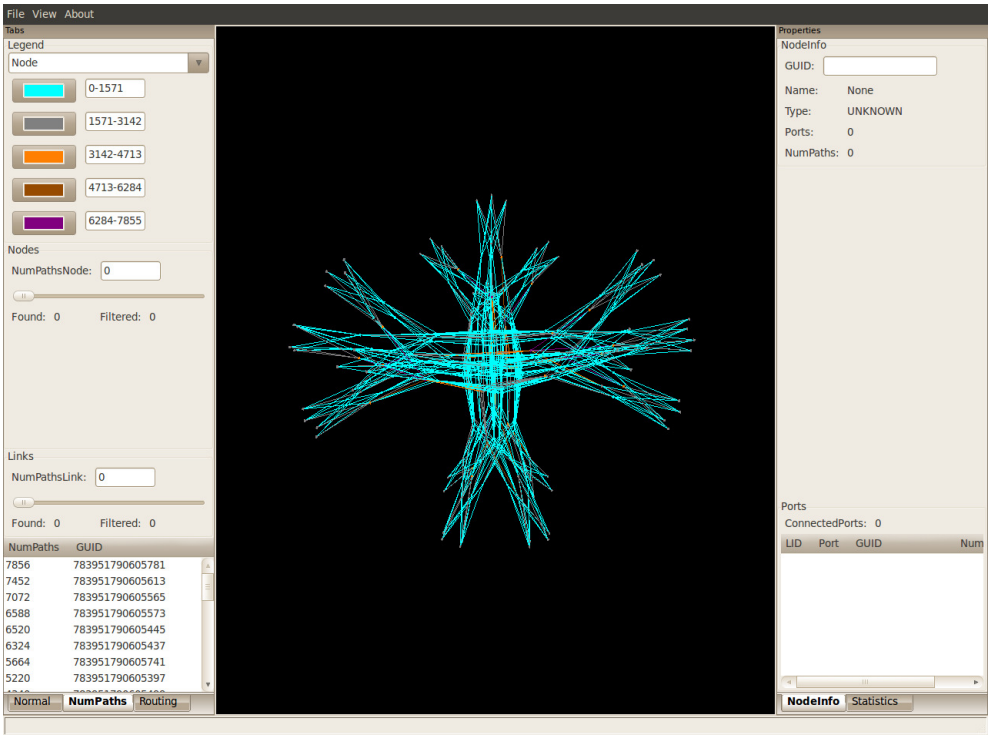


(b) 4-ary 3-tree visualised by IBSimVis, using NumPaths view.

Figure 6.2: 4-ary 3-tree visualised by IBSimVis.

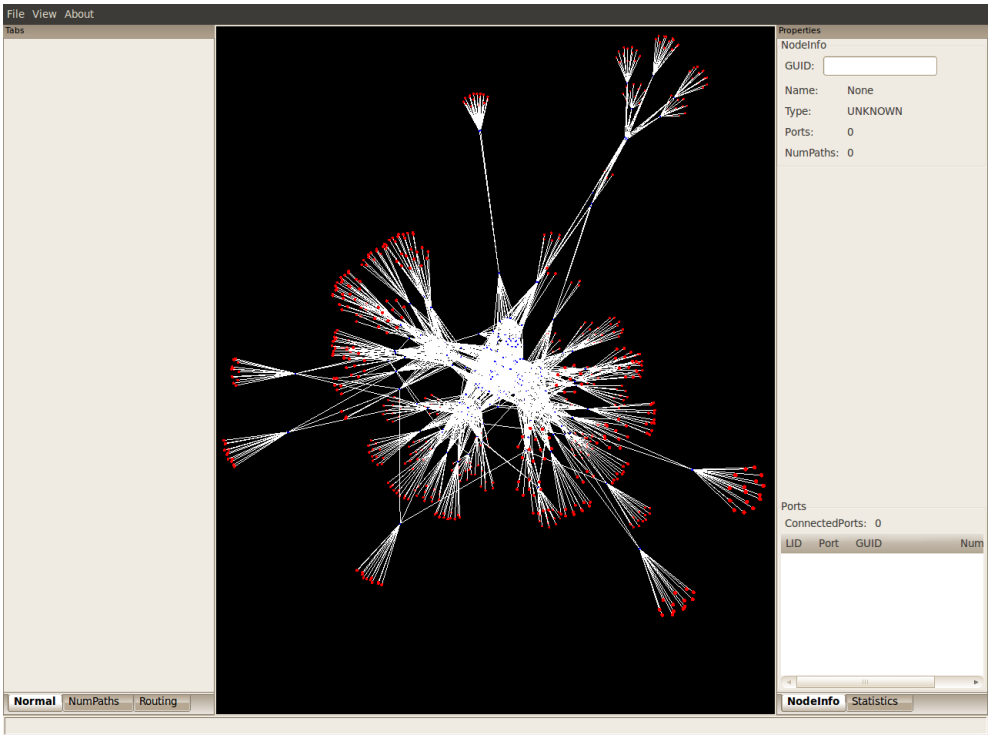


(a) 4-ary 4-tree visualised by IBSimVis, using Normal view.

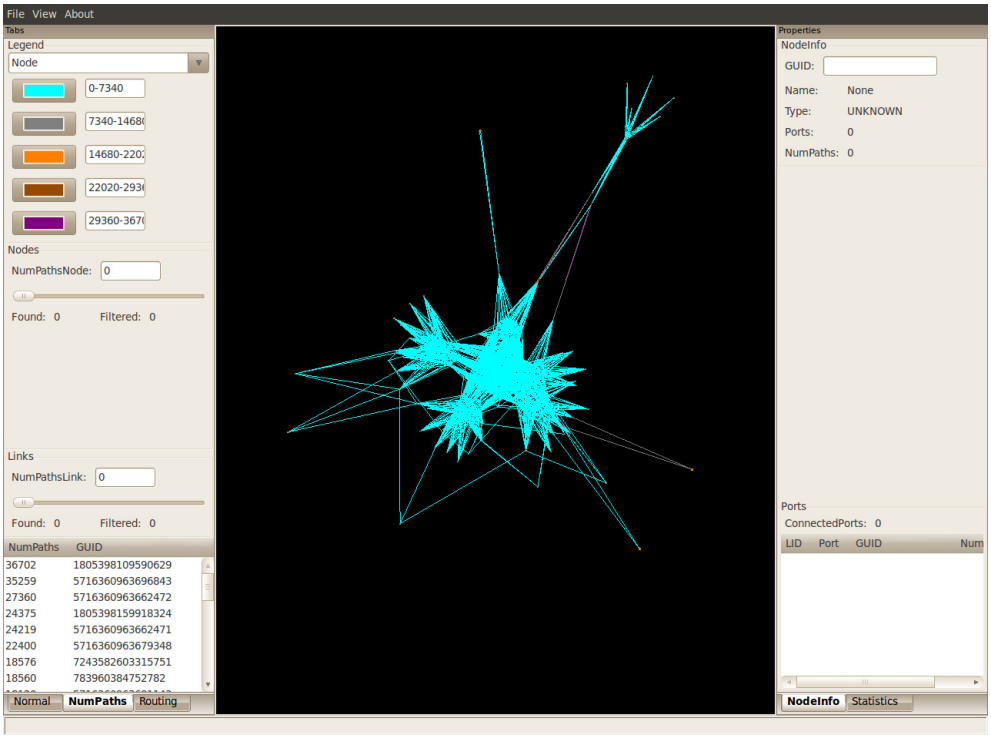


(b) 4-ary 4-tree visualised by IBSimVis, using NumPaths view.

Figure 6.3: 4-ary 4-tree visualised by IBSimVis.



(a) TITAN visualised by IBSimVis, using Normal view.



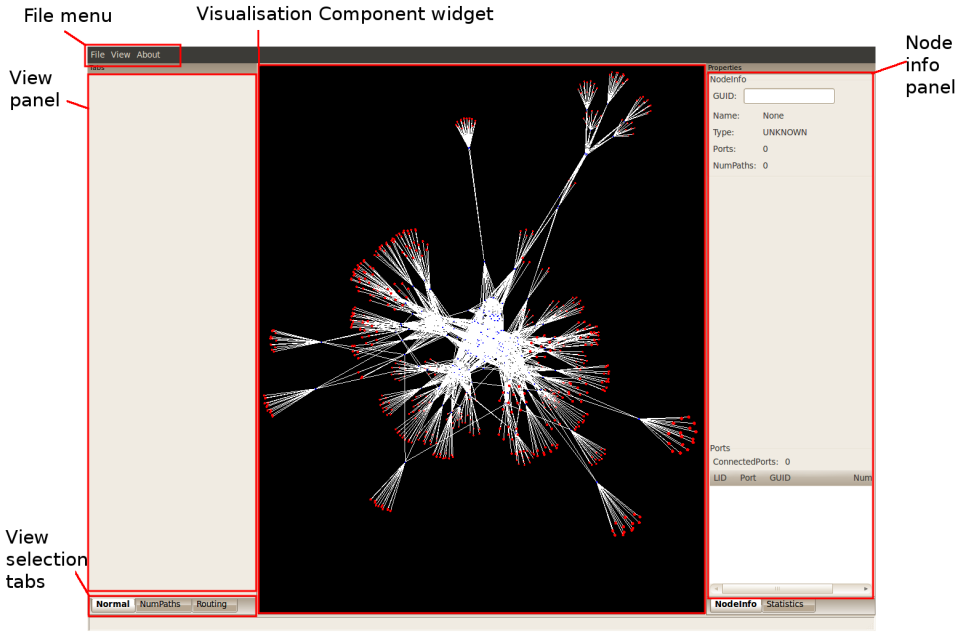
(b) TITAN visualised by IBSimVis, using NumPaths view.

Figure 6.4: TITAN visualised by IBSimVis.

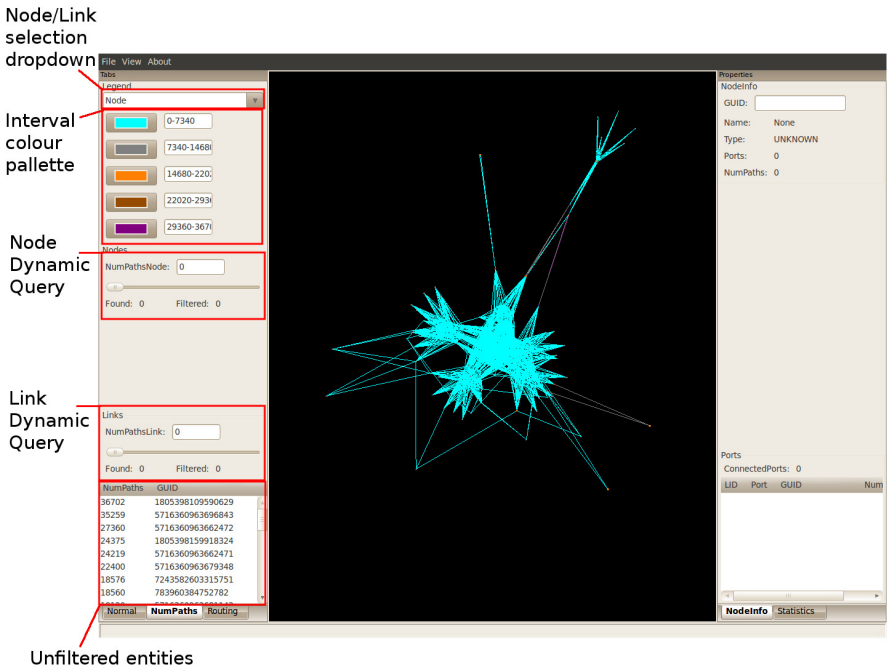
Note that the first impression was not good for the small topology in Figure 6.1a. Originally, it was rendered outside the camera's viewport, so we had to rotate, zoom and pan to locate where it was. This issue was probably due to a bug in the function that was responsible to center the topology. However, this was only in the case of the smallest topology, as the other topologies were centered pretty well, where we only had to zoom abit to fit the entire network topology in the VC widget. The 4-ary 2-tree (Figure 6.1) and 4-ary-3-tree topologies (Figure 6.2) visualised by IBSimVis can be compared to their 2D versions in Figure 2.4b and Figure 2.4c, respectively.

Taking a look at Figure 6.5a and Figure 6.5b, we see how the GUI design turned out. In Figure 6.5a, we see the view panel to the left, where view information is shown. At the bottom left, there are tabs to select the Normal, NumPaths and Routing views, respectively. On the right side we have a node info panel, where the results of our Entity Selection Queries are displayed (currently no node is selected), showing basic node information in addition to its port connectivity. At the bottom right, the user is able to select between the node info panel and the statistics panel, showing basic statistics about the topology such as the number of nodes, switches and CAs.

Viewing Figure 6.5b, we can see the same view panel to the left, but this time with the controls mentioned in the design. Namely the colour pallette chooser and dynamic query sliders for both Link and Node entities. By clicking on the drop-down, one can choose between colouring Node or Link entities. By clicking the coloured button, one can select a different colour. The default colours from the lowest interval to the highest, for both Node and Link entities are: cyan, grey, orange, brown and purple. These colours are in the same colour set recommended by Ware [21] in Section 2.2.1. In addition, at the bottom of the view panel, one can see the the currently displayed node entities and the number of paths going over them.



(a) Showing the main GUI elements of IBSimVis.



(b) Showing the GUI elements of the NumPaths View in IBSimVis.

Figure 6.5: Showing the GUI elements of IBSimVis.

6.2.2 Scaling

As mentioned in Section 4.3.3.1, one of the main problems with the family of graph drawing algorithms we have chosen, is time complexity. Here, we investigate how IBSimVis scales with regards to time as a function of the number of nodes present in the topology. We also see where in the application there was the most overhead, and which components that require optimisation. For this experiment, we ran three tests, where we investigated:

1. How long time it takes to parse and set up the network topology data structure.

2. How long time it takes to parse, network topology creation and run the layout algorithm.

3. How long time it takes for the user to start the application and get the visualisation on screen.

We expect the parsing stage and topology data set-up stage to to take $O(|E|)$ time, since the topology file parses the links in the fabric, that are passed only once. We expect the current implementation of the layout algorithm to take about $O(458|V|^2+|E|$ time, since it uses 458 iterations running attractive and repulsive force calculations using $O(|V|^2+|E|)$ time, per iteration on the topology. We expect the time it takes from parsing, data set-up and layout to take about $O(2|V^2|+|E|)$ time, including the path distribution calculations ($O(|V^2|)$) and the scene creation ($O(|V|+|E|)$.

We performed five runs per test and took the average total time of these, using the UNIX `time` tool. To make IBSimVis stop when we wanted it to stop, we simply used the `exit` system call to terminate the process, when it had done the tasks as specified above. Here, we are only interested in total time, as this is what the user experiences when waiting. We also included two plots per test, one using the BGL and one using LEMON. In Section 5.3.3 we chose the BGL instead of LEMON, since we had performance issues with LEMON. Those results are also included here and it impacted our final choice of graph datastructure library. Our results are shown in Figure 6.6, Figure 6.7 and Figure 6.8.

As seen in Figure 6.6, both implementations used about the same time to parse and set up the datastructures for all the topologies. The time it took for the largest sample topology, was a little over 6 seconds. Note that handling the routing data file for the TITAN topology, requires parsing 1 450 384 lines and there are constraints when it comes to hard-drive access times.

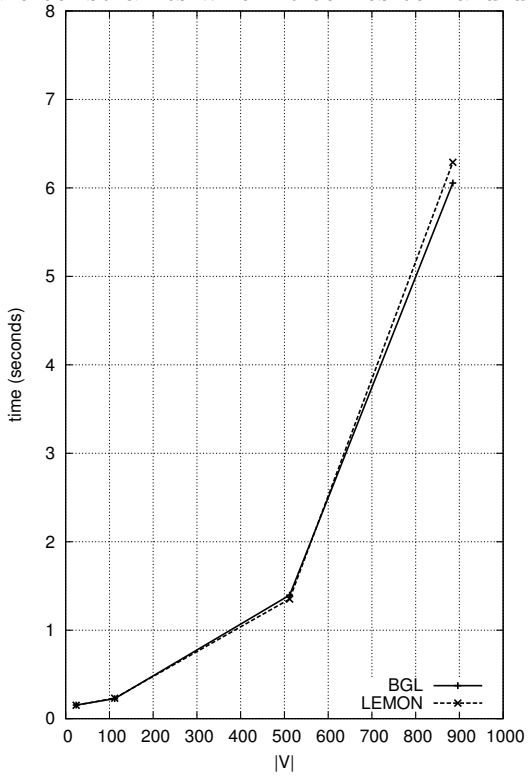


Figure 6.6: Execution time versus the number of nodes in the topology, by running parsing and topopology insertion only.

Figure 6.7 shows the time used by each library to parse, set up data structures and layout the network topology. In the figure, we can see a substantial difference in performance between the BGL and LEMON. BGL is almost 3 times faster using the TITAN topology. Although, waiting 26 seconds using the BGL is still quite a long time for the user to simply wait.

Figure 6.8 shows the time from the user starts IBSimVis, until the first frame of the visualisation has been rendered. From the figure, we can see an overhead of about 3 seconds for the BGL and 2 seconds for LEMON. So the time it takes for the GUI and VC components to draw the visualisation

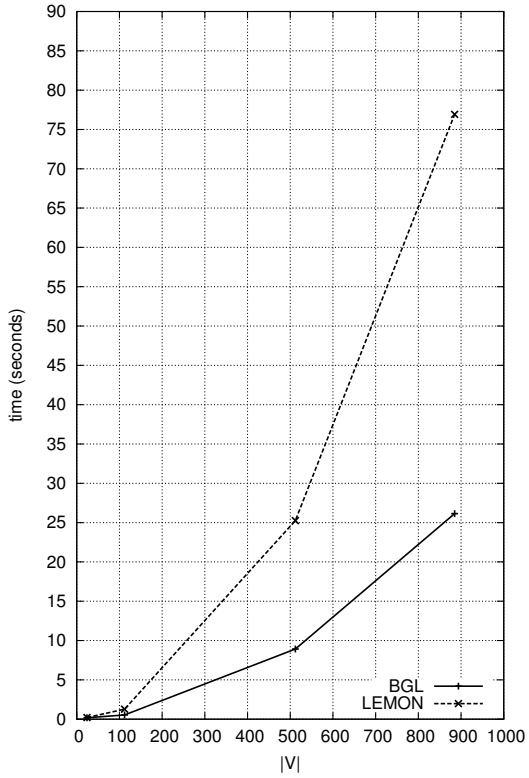


Figure 6.7: Execution time versus the number of nodes in the topology, by running parsing, topopology insertion and layout algorithm.

is minimal compared to the time used during layout. We also observe that the LEMON implementation was faster, which might indicate a flaw in our LEMON implementation, compared to the BGL implementation.

In this experiment we have seen how long time it takes to run various parts in IBSimVis, for two different implementations. We saw that our expectation in regard to the performance of the layout stage was correct, with most of the time complexity tied up in graph layout algorithms. The other stages had far less overhead and this would further be remedied using more modern and faster hardware. Even though the user does know how far along the layout algorithm has come, thanks to the progress bar in the command line, waiting almost 80 seconds to see the visualisation is not good. And this is considering the TITAN topology is of medium size and there does exist much larger topologies than this in simulation settings. However, once the user has the visualisation up and running, there are no more waiting periods.

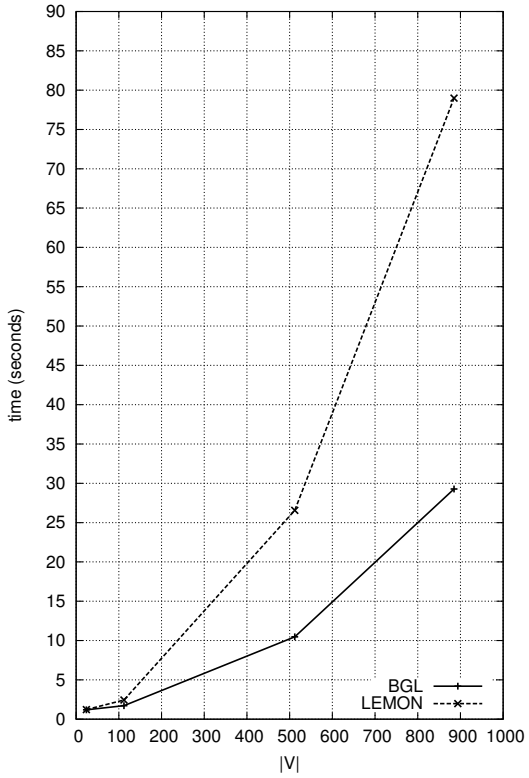


Figure 6.8: Execution time versus the number of nodes in the topology, from the starts IBSimVis, until the first frame is rendered in the visualisation.

Since interactions in data selection and navigation are responsive and immediate. We expect the user to take time to explore the visualiations, once they are up and running. Also, one can not ignore the parsing time required for the data files, especially the routing data file since it has LFT entries for each LID in the fabric. We also assume that the user has a more modern workstation, than the one the tests were run on. Based on our measurements we recommend the tool to be used on medium sized topologies, with maximum 1800 nodes, assuming the user can tolerate waiting for one minute. Note that this recommendation needs to be adjusted, depending on what machines the user runs IBSimVis on.

6.2.3 Usability evaluation and use cases

Here we evaluate the usability of IBSimVis. We compare the presentation against the interactive elements and assess how useable it is. To do this, we

attempt to apply the usability definitions in [89], namely:

Learnability If the system is easy to learn and the user can get it up and running fast.

Efficiency If the system is easy to use so that the user can increase productivity.

Memorability If the system and interface is easy to remember, without the user having to re-learn the system.

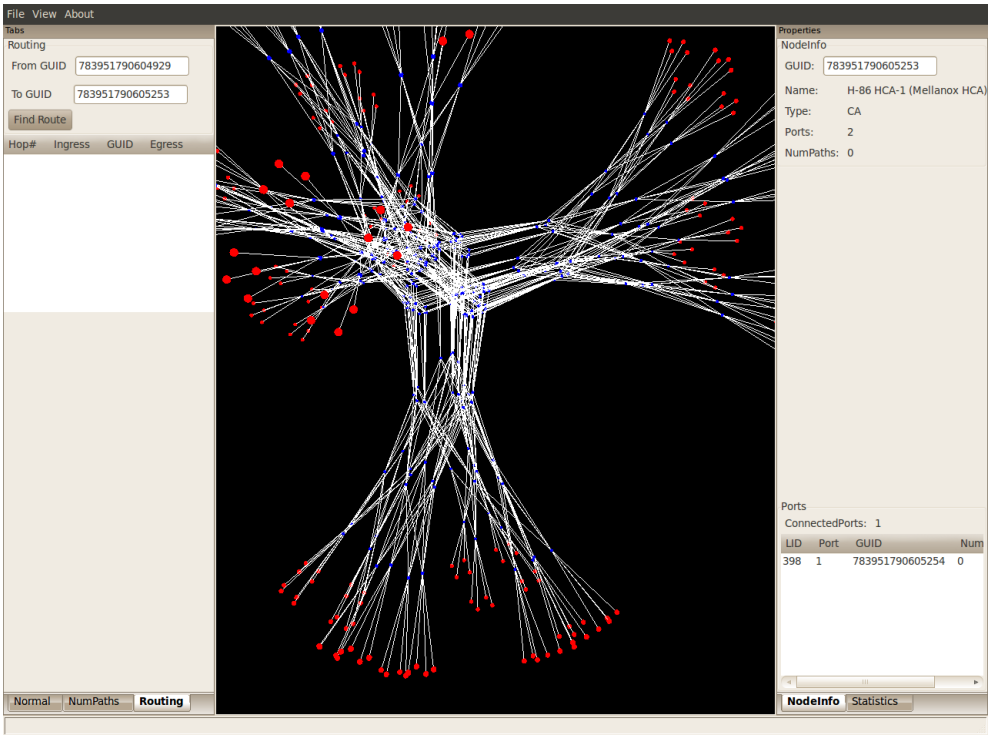
Errors If users of the system can make few errors and if they are able to recover from catastrophic failure.

Satisfaction If the system is subjectively pleasant to use.

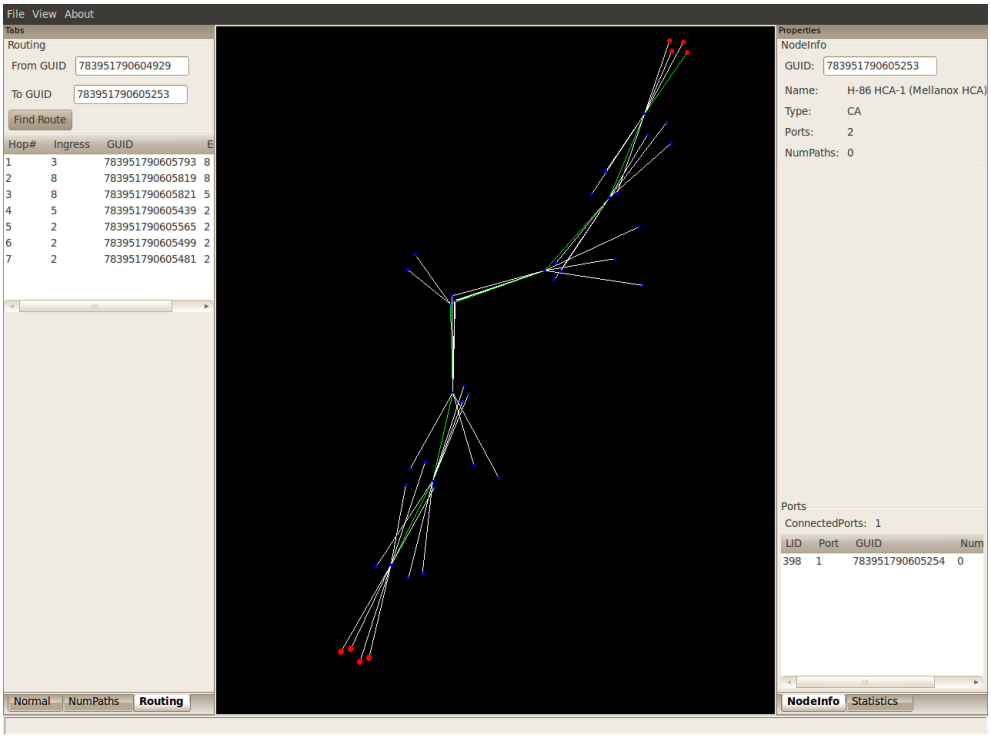
The suggested way of measuring the usability of an application, is to select a representative set of test users to try to solve a set of predetermined tasks [89]. Then allow them to grade the system by use of questionnaires according to the five definitions provided above. Since we did not have time to do a user study, we instead attempt to do use the two views in IBSimVis and evaluate how well they work. We attempt to investigate how our vision and IBSimVis can help gain insight or answers to both fuzzy and concrete questions regarding the network topology or routing data. Next, we evaluate the two views that we believe researchers are going to find useful.

6.2.4 Routing view evaluation

In this use case, we attempt to evaluate the usability of the Routing view in IBSimVis. Loading up the 4-ary 4-tree, shown in Figure 6.9a, we route between two nodes. The choice of these nodes were picked at random and the way they were input, was by copying the GUID text field in the node info panel on the right to the route input boxes. Afterwards the Route button was clicked. The result is shown in Figure 6.9b, as the route is shown in green between the two CAs.



(a) Initial view of the 4-ary 4-tree.



(b) Routing between two nodes in the 4-ary 4-tree

Figure 6.9: Result when using the Routing view.

The positive about the routing view is that a researcher can get immediate information about how the route looks like. Typically the view can be used to answer concrete questions about a certain path the researcher is curious about, for example after having found a bug during simulation.

The negative about the Routing view, is that even though we route from top CA in the figure, to the bottom CA, it is not possible to see which direction the route took. There is also an issue colouring the last hop toward either the source or destination. We also find the view to be confusing, as we have no idea where in the topology we currently are. Perhaps a different technique in obscuring irrelevant links and nodes should have been used. Another drawback to this view is that it has limited use during exploration of a network topology, not being able to answer fuzzy questions. Since the exact GUIDs need to be input in the two input fields.

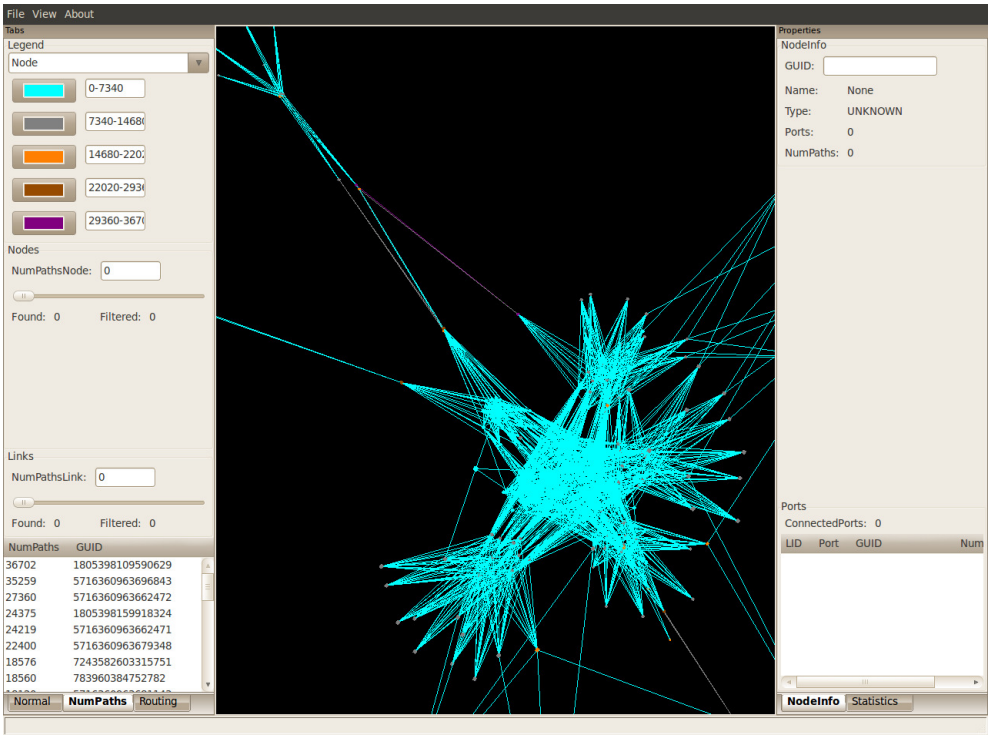
6.2.5 NumPaths view evaluation

In this use case, we see if the NumPaths view can aid researchers in detecting patterns or anomalies. We analyse the TITAN computing cluster topology using IBSimVis and see what the NumPaths view can tell us and how usability affects this.

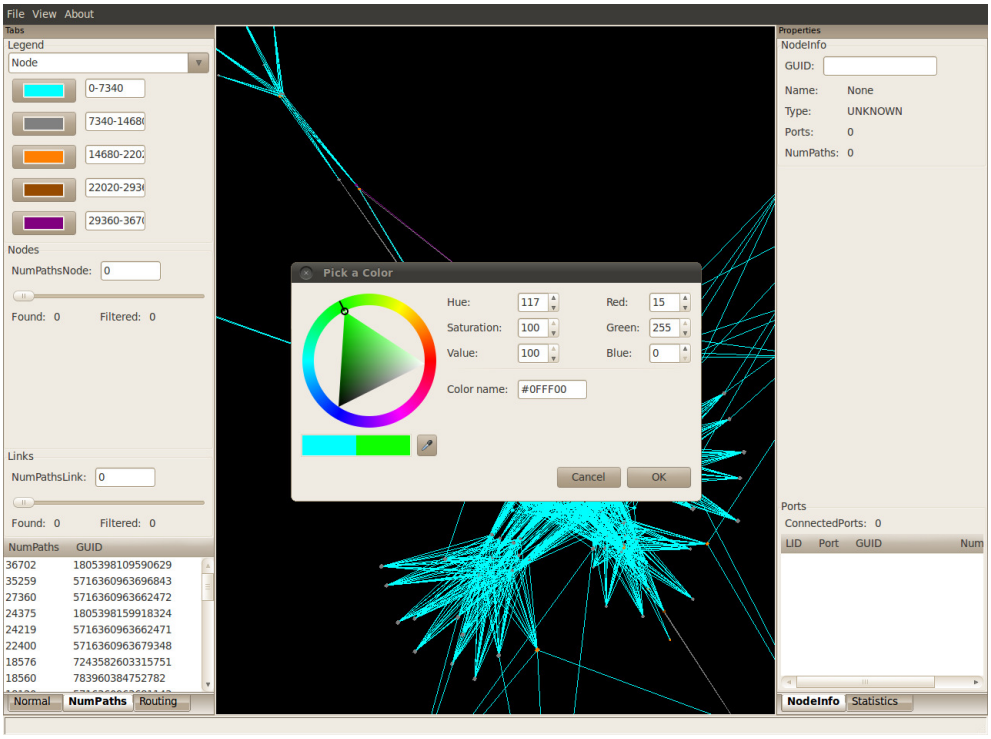
As shown in Figure 6.4a, initially we were greeted with quite a complex graph structure. Except for its pleasing aesthetics, the only thing it told us after zooming, panning and rotating the topology around abit, was the there clearly was a subtree connected to the main cluster. Turning on the NumPaths view in Figure 6.10a, the network topology looked like a cyan ball of tangled string. To better see the cluster structures in the graph, we changed the colour of the lowest interval to a green colour as this contrasts well with the background as shown in Figure 6.10b. But as we can see in Figure 6.10c, it was a poor choice since the topology is dominated by the cyan, which we know is close to green in the RGB colour scale. As shown in Figure 6.10d, we changed the lowest interval link colour to white to help contrast against the green nodes. Now that the graph was more readable, we decided to use the dynamic query filter, to filter out some links to make

the graph more readable as shown in Figure 6.9f. Sliding it some more, an interesting link was discovered immediately and coloured red for emphasis as shown in Figure 6.8h. We noticed that two nodes connected to a link had the highest number of paths in the topology. Indicating a potential bottleneck. Recalling from earlier, we found out that the link in question is connected to the subtree structure. This meant that the two intermediate switches between computing nodes at the end of the subtree and the rest of the topology, could lead to lower performance in the cluster. Identifying the same could also be done in a text-based analysis script, but it would not be possible to discern *where* in the topology this anomaly occured. As seen in Figure 6.4a, the bottleneck between the subtree and main cluster was intuitive to us, enabling us to more efficiently reason as to the cause of this and also how to solve it. This example showed the strength of IBSimVis and information visualisation, being able to visually point out interesting areas in the network topology. Also notice the cluster structures, in the graph when links were filtered out.

The advantages of the NumPaths view is that a user can receive visual information about path distribution in a network topology. It can help prove that a path distribution of a network topology is unevenly distributed. In addition to show where in the structure it occurs. This allows for fast identification of potential bottlenecks. Another advantage is that the NumPaths view works regardless of routing algorithm applied to the topology. It is also customisable, enabling the user to change colours on the path intervals, according to the users own liking. Also taking into account colour blindness. The disadvantages is that it the division into colours offers little accuracy in the visualisation, one has to click an entity or refer to the sorted path distribution list to retrieve the exact number of paths. The flaw in the coloured path intervals is also apparent when studying Figure 6.8h, as almost all links are filtered out with the dynamic query slider for links only halfways. This means that the vast majority of links are categorised using the lower four colour intervals, if there is even one link or node with high path number value. This can lead to potentially interesting links being pushed to the background visually, as IBSimVis deemed the links uninteresting.

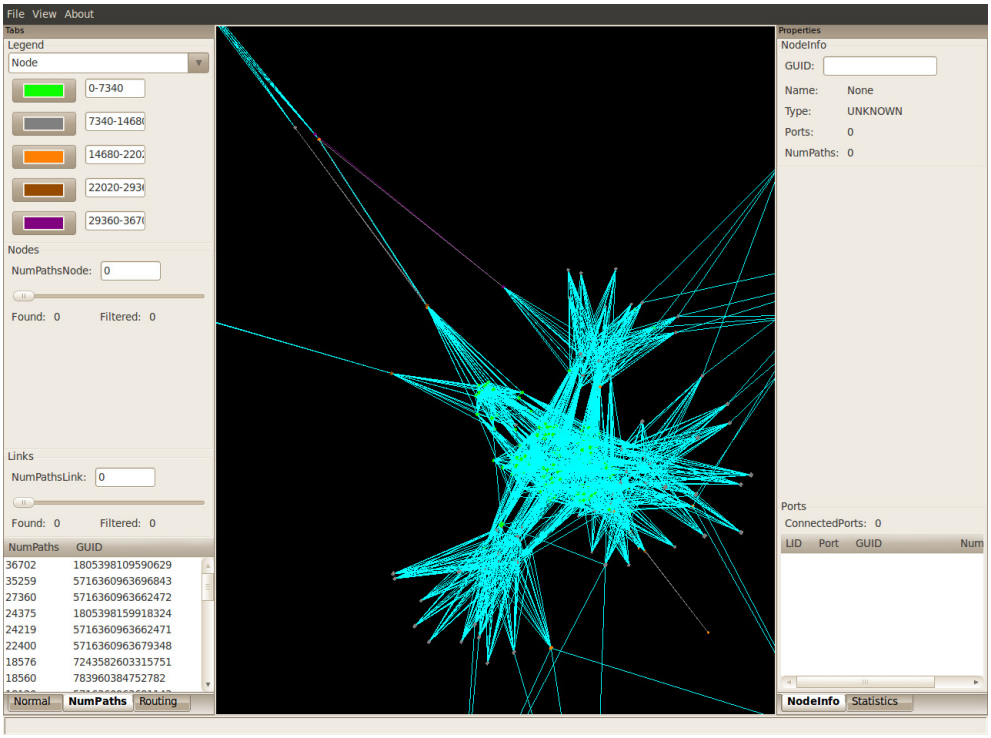


(a) Initial layout, after swapping to NumPaths view.

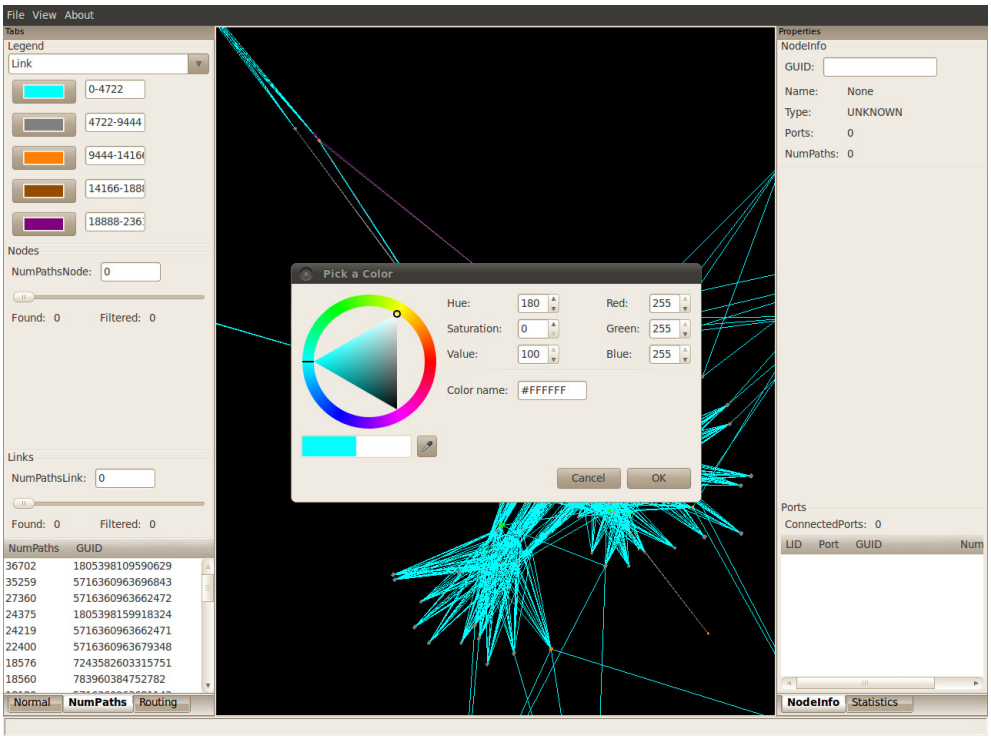


(b) Changing colour on the lowest interval nodes, in an attempt to make them more distinct.

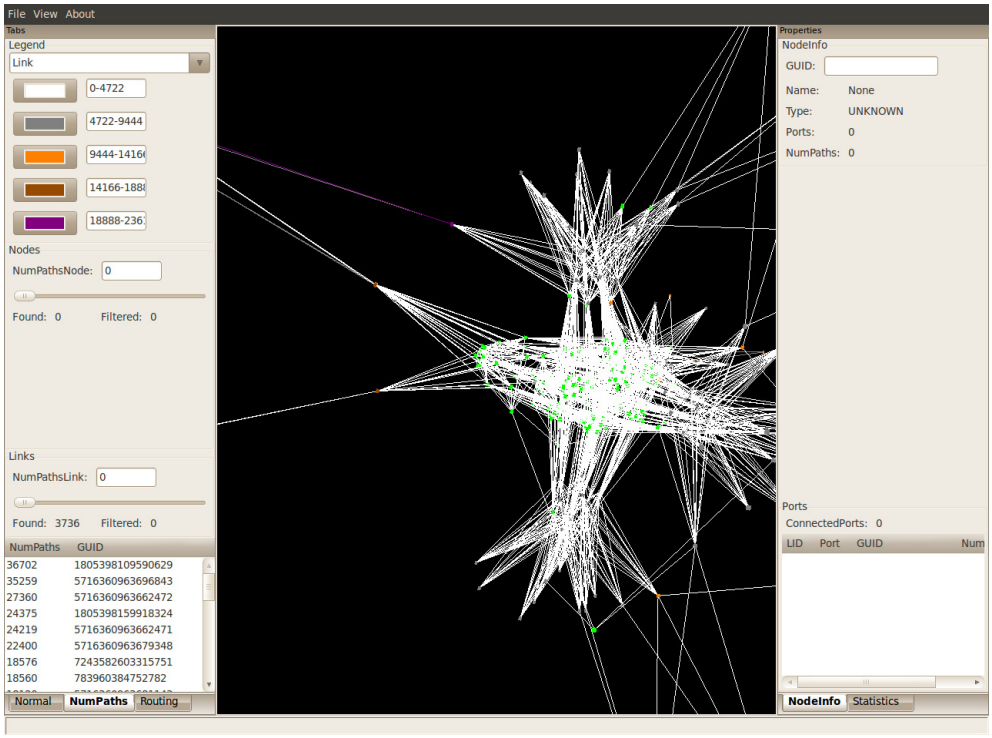
Figure 6.10: Result when using the NumPaths view.



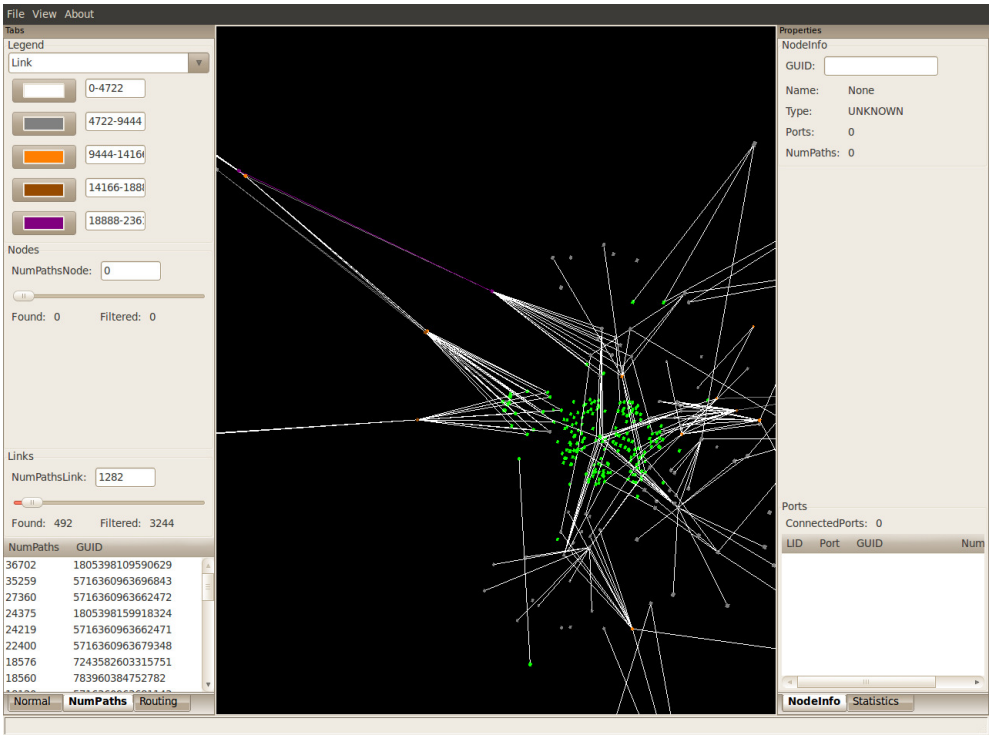
(c) Choosing a green colour for the lower interval nodes was a bad choice, as can be seen.



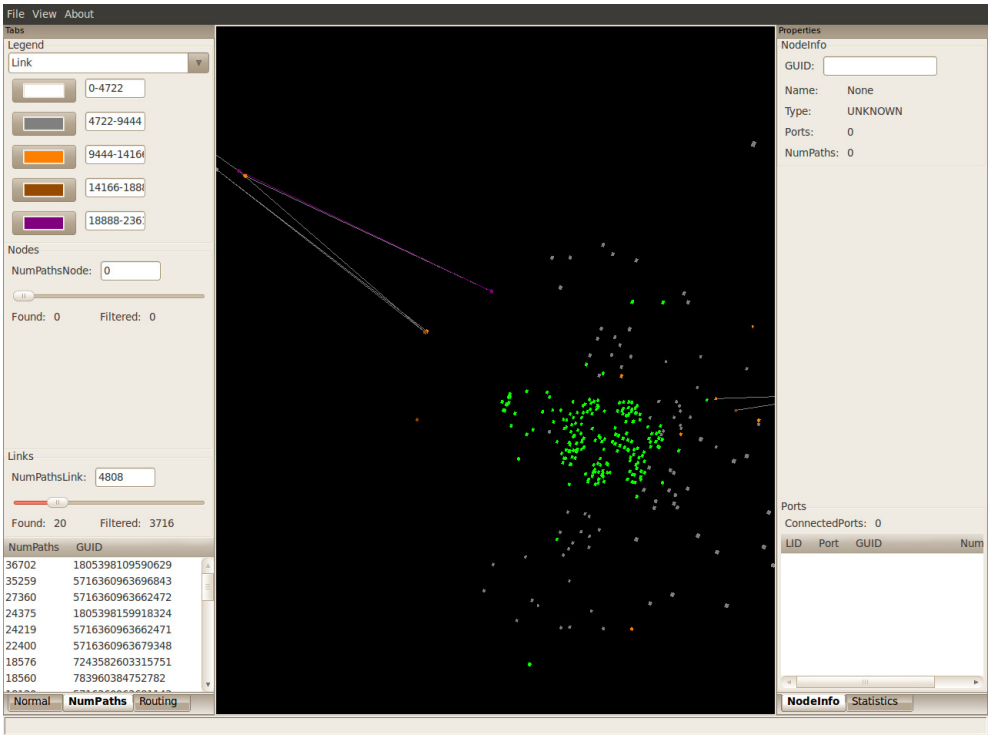
(d) Attempting to change lowest interval link colour to white, to better see the cluster structures.



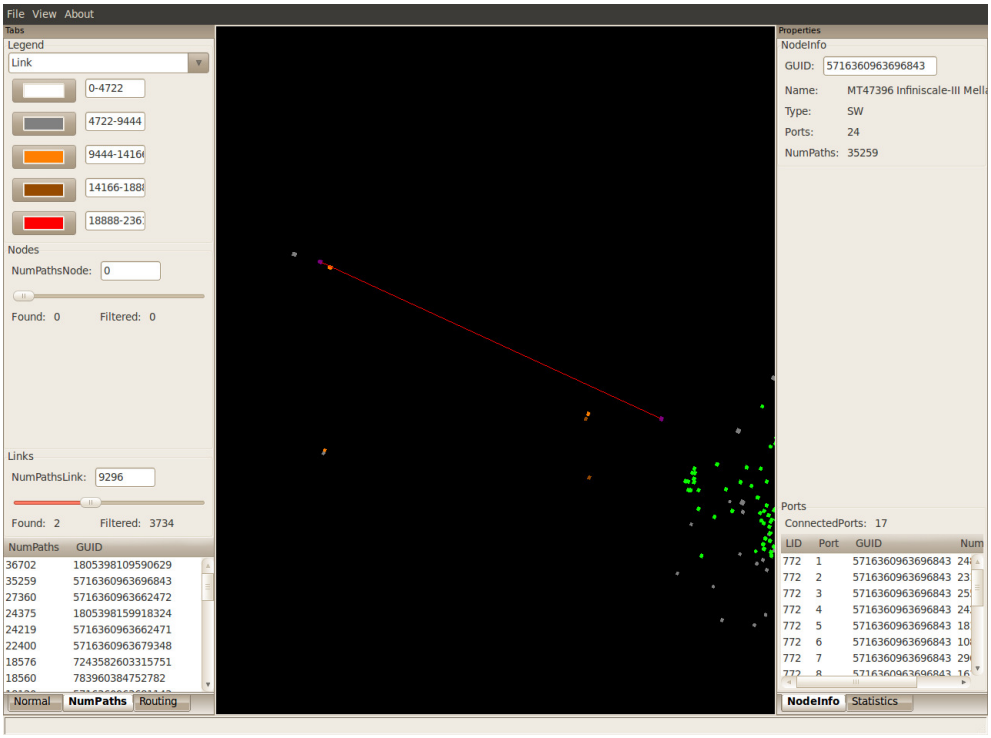
(e) The green nodes on white links gives good contrast, but there is still alot of occlusion going on.



(f) Starting to filter out nodes by number of paths, in order to see better.



(g) Almost all links filtered out, where some links stand out.



(h) Changed colour to red on the highest interval links and navigated to get a better look.

6.2.6 Usability discussion

Here we provide a discussion on each usability topic.

Learnability

As seen in the first impression and the two use cases, IBSimVis features a simple GUI with few options for interaction. The overhead of starting to explore the topology and analyse it should be low. Simply using the left mousebutton rotates the camera around the model, right clicking for pan and using the mouse wheel for zooming. Left clicking a node entity selects it and display information. As is evident, IBSimVis is not a complex application on the surface, even though it does offer complex solutions under the hood. We believe that IBSimVis is easy to learn, both for novices and experts in the research fields surrounding interconnect topologies and routing.

Efficiency

As mentioned, IBSimVis offers a simple set of interaction options for navigation. However, navigating to get a good view of the network topology depends on how large the topology is and how it is drawn. As previously mentioned, small topologies had a chance of being rendered outside the camera viewport, thus we had to rotate around to locate it and then clunkily use a combination of pan, zoom and rotation to get a good view. But exploring the topology was hard to do, since the pivotnode that the camera rotates around, was still anchored at the center of the scene and not the center of the topology. For large topologies however, navigating was usually done within seconds. The only noticeable drawback was the slow scroll speed, if a topology used too much space.

Selecting nodes worked out nicely, although there could have been additional visual cues to show if the node actually was selected. Even though one cannot select links using the mouse in IBSimVis, there could still be interesting links that we want to know more about. To get information about such links, one has to first click on the node incident to the link in question, then the node on the other side of the link. Afterwards identify the port and

see if it leads to the other node. This way of identifying information about links is cumbersome, so we propose that this ability should be of priority in a future version of IBSimVis.

Using the default colour set for both Node and Link entities, for the same intervals proved troublesome. Where we had to find out which colours to use ourselves, initially choosing unsuitable colours in the NumPaths example. Although changing colours to suit the users needs is easy to do, by just simply clicking a button in the GUI.

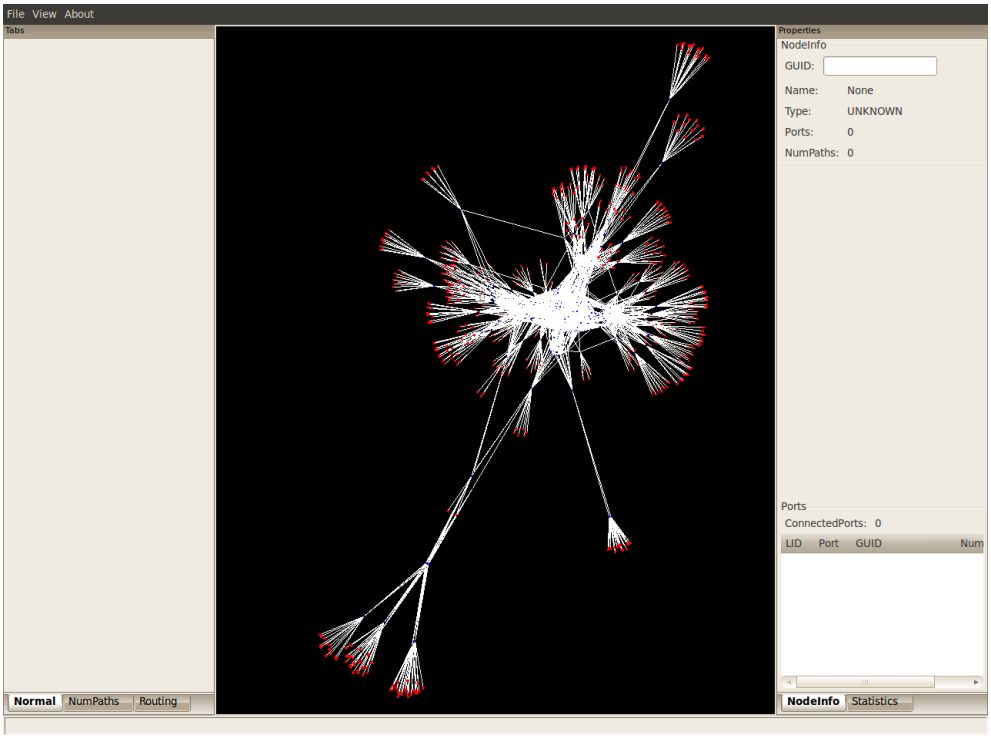
A key question regarding effiency is how IBSimVis would perform compared to a script that parsed the same input data, and analysed for example path distribution. We believe, because of the delay in layout algorithms that a script could find the raw numbers faster and more efficient. However, as mentioned earlier the numbers would give little clue to where in the network topology a possible choke point was or give an overview of how the paths in the rest of the network was distributed. In addition, the dynamic queries worked out really well, being responsive and able to smoothly filter out nodes and links. For these reasons, we believe that IBSimVis is easy to use, being able to cut down on the time a researcher would come to the same conclusion from textual output. In addition, IBSimVis can help on finding answers to fuzzy questions regarding a topology, just by navigating it.

Memorability

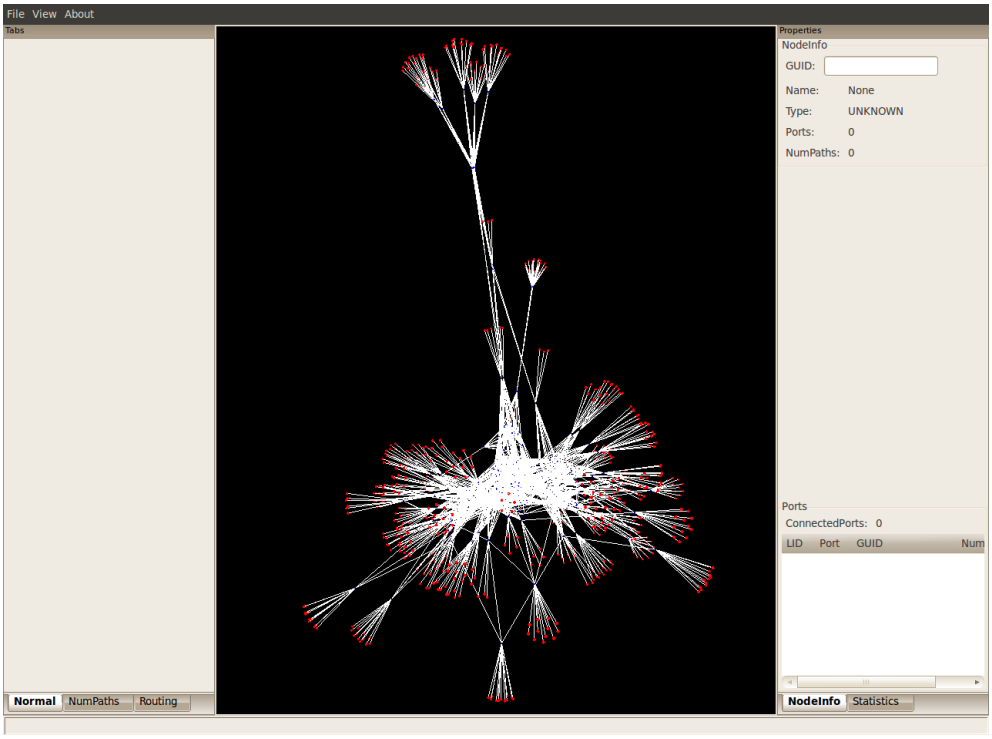
IBSimVis has few options for interaction and the GUI shows what we believe is a modern style, with file menus and context panels on each side of the VC widget. This is similar in style to the general graph analysis tools we have mentioned earlier, such as Gephi and Cytoscape. We argue that a user should easily be able to remember how to interact with IBSimVis, even after not having used it for a while, due to its simplicity of design and clear GUI.

However, when it comes to the network topology visualisation, it is an entirely different matter. Predictability is a term we have used before in this thesis, to define whether a network topology is recognisable from one visualisation run to the other. Since we have a used an unpredictable lay-

out algorithm, this issue is mostly noticeable on the TITAN topology, since it is irregular. For clarity, we have attached Figure 6.8i and Figure 6.8j for comparison. This is just visualising the TITAN topology, zooming out and panning to fit the topology in the VC widget. Notice how the subtree protuding from the main cluster is angled differently. This means that the user gets a new layout to re-learn each time the application is run on the same topology. For regular topologies, this has not been as much as an issue however. This issue supports our idea of having a layout file format as discussed in the design, or alternatively find layout algorithms that are more predictable for irregular topologies.



(i) First visualisation



(j) Second visualisation

Figure 6.8: Comparing the predictability of the TITAN topology.

Errors

Due to the few features in IBSimVis, both in terms of interaction and views, the user is not able to do many errors. In case the user tries to do an illegal action, for example routing from a switch to a switch, there are error handling mechanisms to prevent catastrophic failure for that. Errors however, may occur if the user inputs topology or routing data files in the wrong formats to IBSimVis, resulting in an unrecoverable catastrophic failure. This is also true if the file formats are of the wrong version of OpenSM currently supported.

Satisfaction

We found using IBSimVis was a satisfying experience, after it had started up. The visual aspect, with the use of 3D has definitely an aestheticly pleasing appeal. Especially the regular topologies look good, since they show symmetric properties.

6.3 Summary

In this chapter, we have shown the visual aspects of IBSimVis and provided both a quantitative and qualitative evaluation of usability in regard to visualising static network topologies and routing data. First we gave a first impression look at IBSimVis, where we showed and described the visual features of both the GUI and VC widget.

We then moved on to do a series of experiments, evaluating how the different components in IBSimVis scales in regard to the number of nodes present in the topology. Where we pointed out that layout algorithm computation time is the achilles heel in our application, greatly increasing startup delay. We also saw how our implementations using the BGL and LEMON respectively, performed with BGL being the clear choice. Assuming the user is willing to wait a minute for a visualisation, we recommended that the upper bounds of topology size in IBSimVis should be around 1800 nodes. We argue that this is enough to visualise medium sized InfiniBand clusters. In the future we would like to finish the multilevel layout algorithm implementation

to speed up layout time.

Finally we did a qualitative evaluation of the usability of IBSimVis and examples in the form of use cases, how IBSimVis can be used by researchers to create the network topologies and routing algorithms of the future. In the end, it gives researchers the ability to evaluate for example routing algorithms by viewing the path distribution of the topology. Obvious anomalies such as links or nodes with a high load stands out to our visual system, requiring no concious thought, thanks to preattentive processing.

Looking at the results presented in this chapter, we believe that IBSimVis, with the suggested improvements, is a useful tool to aid researchers in creating and evaluating network topologies and routing data. By enabling them to "use vision to think".

Chapter 7

Conclusion

In this thesis, we have presented a design and a visualisation tool prototype, named IBSimVis used to visualise OpenSM-defined network topologies and routing data in simulated InfiniBand fabrics. The scope of this thesis was limited in the application implementation to only cover static data and not simulation data, due to both time constraints and implementation challenges.

We evaluated IBSimVis, both in regards to scalability and usability. We managed to visualise the network topology, routing data and path distribution using OpenSM data files. We managed to apply interesting techniques in information visualisation literature to create IBSimVis. In addition, we also related the process of information visualisation to IBSimVis's program flow. We successfully used the field of graph drawing to present aesthetically pleasing layouts of both regular and irregular network topologies, at the cost of startup time and predictability. However, we only partly managed to apply a layout algorithm to our visualisation, instead reverting from a multilevel layout algorithm to a single-level layout algorithm. We managed to overall create a visually pleasing view of a network topology and path distribution over an InfiniBand fabric in 3D. In turn this enables researchers to quickly get an overview of a network topology and identify patterns and anomalies in path distribution from routing algorithms.

In the end, we believe our thesis work has resulted in a useful visualisation tool prototype for visualising static properties in simulated InfiniBand

subnets. Its main strength being its NumPaths view for visualising path distribution over a network topology. And its main weakness being that it has a long startup time due to a compute-intensive implementation of a layout algorithm. Our vision for a useful visualisation have only been partly fulfilled and we believe we also have included design considerations that can be of use when further extending IBSimVis.

7.1 Further work

As mentioned earlier in this thesis, we have considered many improvements to IBSimVis, that are interesting for further study. We summarise that further work here, by reflecting on a future vision of IBSimVis.

Visualising simulation data

Since we in this thesis work, did not manage to visualise simulation data, that topic is an obvious candidate for further investigation. As mentioned in the design, we outlined some challenges for parsing simulation data. In the future it would be interesting to see how parsing simulation data would be handled. Not only in regards to the actual parsing itself but also how to able the visualisation to generate the whole picture and grouping events around messages, from production to consumption. Regarding parsing, we mentioned the use of more advanced parsing techniques using the flex and Bison toolchain. This would make IBSimVis more flexible and let it be extendable not only to read event log simulation traces. We also proposed the idea of a general file format, which would be exciting to define. It could either be a human-readable text format or a more effective binary format, containing both static and dynamic data.

In our design we only took into account the visualisation of topology and packet transmission, although we did not have time to implement the latter. Another interesting future extension would be to cover all aspects of a simulation run. For example, showing buffers of messages filling inside the nodes, where a researcher can quickly see the reasons for congestion and see

how it potentially spreads in the network.

Optimisation

As we saw in our result, IBSimVis was hampered by a slow startup thanks to the time consuming variant of a layout algorithm we implemented. Another obvious improvement would be finish the multilevel layout algorithm implementation, finding a faster and better layout algorithm or find a parallelisable layout algorithm. This would also help in scaling IBSimVis to larger topologies, as it currently is recommended used with medium sized topologies. Another interesting feature would be a stop button for layout generation and animating when the layout converges toward the equilibrium configuration. Allowing the user to decide when the layout is good enough.

Layout

Evaluating the aesthetic qualities of a network topology is a completely subjective opinion, therefore we propose adding different high-quality layout algorithms, so the user can choose by using the GUI. Adding to that, we saw that tweaking layout constants could help us produce better layouts, at the cost of time complexity. We could give the user the option to configure such aspects via the GUI or prior to IBSimVis initialisation.

An approach to also reduce the time complexity could be to animate the network topology for each iteration, as it converges and include a "stop" button. Clicking the stop button would halt the layout algorithm, allowing the user to decide for himself when the network topology looks pleasant. The network topologies produced by IBSimVis were also quite unpredictable, in the future we would like see a layout file format or layout cache to save layout computation when viewing the same network topology multiple times.

3D Visualisation

The use of 3D enabled us to create some exciting and diverse visualisations of network topologies, both regular and irregular in nature. However, we feel that there is plenty of potential left to extend IBSimVis with the more

powerful features of 3D graphics. For example, by adding more visually appealing glyphs using lighting and shaders, in addition to particle effects. IBSimVis is also in need to improvements in regards to for example centering the network topology correctly in front of the camera. Coming back to the ability to view messages in buffers earlier, adding semantic zoom would be exciting. As mentioned in the design, this means that one can zoom in on switches and see their inner workings. Or zoom out on the network topology to reveal clustered nodes.

Visualising routing subgraphs. For example by routing from one source node to all other destination nodes, one can visualise the subgraph and see which switches are not in any path from that source node. A similar feature could be used, by showing nodes that do not have paths over them at all, and thus become redunant, unless a certain degree of redundancy a part of it.

Client-server model

Another interesting feature would be to create a client-server communication model between the DMC and VC components. Doing this, one can possibly run visualisation on a computer dedicated for that, with a computer dedicated for fast parsing and computations pushing only visualisation changes to the visualisation. This would also mean a loose coupling between the implementations of VC and DMC, allowing for more flexibility in the future.

IBSimVis as plugin to OMNeT++

As of now IBSimVis is a stand-alone application, but future work could include integrating the VC into OMNeT++. For example by creating a IBSimVis OMNeT++ plugin or create a fork of OMNeT++ specialised for InfiniBand simulation visualisation. This would also give the option to add real-time support, enabling a researcher to see not only the path distribution, but also how it actually works out in simulation scenarios. Adding a timeline for playback and recording functionality would also be a useful feature, in line with visualisation tools we have reviewed.

Cross-platform compatibility

Currently IBSimVis only works on the Linux platform, since some users may prefer other platforms, we would like it to be extended to reach those other platforms aswell. This is made easy, since all the dependencies in IBSimVis, has support for the most common platforms.

Statistical data analysis tools

IBSimVis could also be extended to calulcate statistical data, such as being able to aggregate, sum or average the number of package per node, per link, per port, per buffer and so forth.

Real-time surveillance

In the end, real-time surveillance of InfiniBand networks would be the ultimate goal for a visualisation tool. Both being able to work with simulations and connected to a real, working InfiniBand cluster.

Appendix A

Source code and documentation

The source-code and documentation of IBSimVis can be found in a tarball at `http://heim.ifi.uio.no/~joakibj/master/`. Please refer to the README file in the distribution for instructions on how to resolve dependencies, build and run IBSimVis. Both BGL and LEMON implementations are included in the distributions.

A.0.1 Sample source code

Listing A.1: Our single-level implementation of Walshaw's multilevel force-directed placement algorithm.

```
1  #define CONST_C 0.2
2
3  void MLFDP::layout()
4  {
5      boost::graph_traits<Graph>::vertex_iterator viv, vi_endv,
           nextv;
6      boost::graph_traits<Graph>::vertex_iterator viu, vi_endu,
           nextu;
7      boost::graph_traits<Graph>::out_edge_iterator oeiu,
          oei_endu, oei_nextu;
8
9      Graph* g = topology->getGraph();
10
11     int i = 0;
```

```
12      double t = initial_k_springlength(topology->getGraph());
            //temperature
13      double tolerance = 0.01;
14      double k = t;
15
16      Graph *Posn = topology->getGraph();
17      Graph OldPosn;
18
19      Ogre::Vector3 theta;
20      Ogre::Vector3 delta;
21
22      int converged = 0;
23      while (converged != 1)
24      {
25          converged = 1;
26
27          boost::tuples::tie(viv, vi_endv) = vertices(*Posn);
28          for (nextv = viv; viv != vi_endv; viv = nextv)
29          {
30              v_desc vdv = *nextv;
31
32              theta = Ogre::Vector3();
33
34              // global repulsive forces
35              boost::tuples::tie(viu, vi_endu) = vertices(*Posn
                    );
36              for (nextu = viu; viu != vi_endu; viu = nextu)
37              {
38                  if (nextu != viv)
39                  {
40                      v_desc vdu = *nextu;
41                      delta = (*Posn)[vdu]->pos - (*Posn)[vdv
                            ]->pos;
42                      theta += (delta / delta.length()) *
                            FR_repulsive(delta.length(), 1 ,k);
43
44                  }
45                  ++nextu;
46              }
```

```
47
48              // local attractive forces (loop over edges and
                    calculate displacement on vertices at either
                    side of the edge)
49              boost::tuples::tie(oeiu, oei_endu) = out_edges(
                    vdv, *Posn);
50              for (oei_nextu = oeiu; oeiu != oei_endu; oeiu =
                    oei_nextu)
51              {
52                  e_desc edu = *oei_nextu;
53                  v_desc vdu = target(edu, *Posn);
54
55                  delta = (*Posn)[vdu]->pos - (*Posn)[vdv]->pos
                        ;
56                  theta += (delta / delta.length()) *
                        FR_attractive(delta.length(), k);
57
58                  ++oei_nextu;
59              }
60              // reposition
61              Ogre::Vector3 reposition = (*Posn)[vdv]->pos + (
                    theta / theta.length()) * std::min(t,
                    static_cast<double>(theta.length()));
62              Ogre::Vector3 oldPosition = (*Posn)[vdv]->pos;
63              (*Posn)[vdv]->pos = reposition;
64
65              delta = (*Posn)[vdv]->pos - oldPosition;
66
67              if (delta.length() > (k * tolerance))
68              {
69                  converged = 0;
70              }
71              ++nextv;
72          }
73          t = cool(t);
74          i++;
75      }
76  }
```

Listing A.2: Repulsive, attractive and cooling functions in our single-level implementation of Walshaw's multilevel force-directed placement algorithm.

```
1  double MLFDP::FR_repulsive(double x, double w, double k)
2  {
3      return (-CONST_C * w  * pow(k, 2)) / x;
4  }
5
6  double MLFDP::FR_attractive(double x, double k)
7  {
8      return pow(x, 2) / k;
9  }
10
11 double MLFDP::cool(double t)
12 {
13     // lambda
14     return 0.99 * t;
15 }
```

Listing A.3: Implementation of the path distribution discovery algorithm.

```
1  void Topology::routeAll()
2  {
3      for(auto iterSrc = nodeMap.begin(); iterSrc != nodeMap.
           end(); iterSrc++)
4      {
5          if(iterSrc->second->getType() == IB_CA_NODE)
6          {
7              for(auto iterDst = nodeMap.begin(); iterDst !=
                   nodeMap.end(); iterDst++)
8              {
9                  if(iterDst->second->getType() == IB_CA_NODE)
10                 {
11                     if(iterSrc != iterDst)
12                     {
13                         route(iterSrc->second, iterDst->
                               second->getPort(1)->getLid());
14                     }
15                 }
16
17             }
18         }
19     }
20 ...
21 }
```

Listing A.4: Routing from source node to destination LID.

```
1  void Topology::route(Node* src, uint32_t dstLid)
2     uint32_t loopCounter = 0;
3      std::vector<Node*> path;
4      for(unsigned int i = 1; i <= src->ports.size(); i++)
5      {
6          if(src->getPort(i) != NULL)
7          {
8              path.clear();
9              loopCounter = 0;
10             Node* cur = src->getPort(i)->getRemoteNode();
11
12             while(cur->getType() != IB_CA_NODE && loopCounter
                   < 20)
13             {
14                 loopCounter++;
15                 path.push_back(cur);
16                 Port* nextPort = cur->getRoutingNextPort(
                       dstLid);
17                 if(nextPort != NULL)
18                 {
19                     nextPort->visit();
20                     cur = nextPort->getRemoteNode();
21                 }
22                 else
23                 {
24                     break;
25                 }
26                 if(cur == NULL)
27                 {
28                     break;
29                 }
30             }
31             for(unsigned int i = 0; i < path.size(); i++)
32             {
33                 path[i]->visit();
34             }
35         }
36     }
37 }
```

Listing A.5: Our implementation of the Entity Selection Query.

```
Ogre::MovableObject* wxOgreFrameListener::selectedObject(
    uint32_t queryMask, wxMouseEvent &evt)
{
    if(lastSelectedObject != NULL)
    {
        lastSelectedObject->getParentSceneNode()->
            showBoundingBox(false);
    }

    Ogre::Ray curMouseRay = mCamera->getCameraToViewportRay(
        evt.GetX()/float(windowWidth), evt.GetY()/float(
        windowHeight));
    mRaySceneQuery->setRay(curMouseRay);
    mRaySceneQuery->setSortByDistance(true);
    mRaySceneQuery->setQueryMask(queryMask);
    Ogre::RaySceneQueryResult& result = mRaySceneQuery->
        execute();
    auto iter = result.begin();

    if(iter != result.end() && iter->movable != NULL && iter
        ->movable->isVisible())
    {
        iter->movable->getParentSceneNode()->showBoundingBox(
            true);
        return iter->movable;
    }
    return NULL;
}
```

Listing A.6: The `newLine` function in `OgreShapeFactory`, showing the OpenGL-like syntax when creating `ManualObjects`.

```
void OgreShapeFactory::newLine(Ogre::ManualObject *obj,
                        Ogre::Vector3 from,
                        Ogre::Vector3 to,
                        Ogre::ColourValue color)
    {
        obj->setDynamic(true);
        obj->setQueryFlags(LINK_FLAG);
        obj->begin("BaseWhiteNoLighting", Ogre::
           RenderOperation::OT_LINE_LIST);

        obj->position(from);
        obj->colour(color);

        obj->position(to);
        obj->colour(color);

        obj->end();
    }
```

Bibliography

[1] Top500 supercomputing sites. `http://www.top500.org`, 11 May 2011.

[2] InfiniBand™Trade Association. *InfiniBand™Architecture Specification Release 1.2, Volume 1*, revision 1.2 edition, 2004.

[3] The network simulator - ns-2. `http://www.isi.edu/nsnam/ns/`, June 2010.

[4] A. Varga et al. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, pages 319–324, 2001.

[5] Omnet++ community site. `http://www.omnetpp.org`, February 2011.

[6] S.K. Card, J.D. Mackinlay, and B. Shneiderman. *Readings in information visualization: using vision to think*. Morgan Kaufmann Publishers, 1999.

[7] D. Estrin, M. Handley, J. Heidemann, S. McCanne, Y. Xu, and H. Yu. Network visualization with nam, the vint network animator. *Computer*, 33(11):63–68, 2000.

[8] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. In *International AAAI Conference on Weblogs and Social Media, North America, mar. 2009 (ICWSM09)*, 2009.

[9] O. e. a. Shannon, Markiel. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome Research*, 13(11):2498–2504, 2003.

[10] M. Batagelj. Pajek - program for large network analysis. *Connections*, 21(2):47–57, 1998.

[11] A.S. Hornby, editor. *Oxford Advanced Learner's Dictionary*. Oxford University Press, 8th edition, 2010.

[12] William James Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2004.

[13] Lionel M. Ni José Duato, Sudhakar Yalamanchili. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers, 2003.

[14] Fabrizio Petrini and Marco Vanneschi. k-ary n-trees: High performance networks for massively parallel architectures. In *In Proceedings of the 11th International Parallel Processing Symposium, IPPS'97*, pages 87–93, 1997.

[15] Top500 supercomputing sites - 2007 overview of recent supercomputers. `http://www.top500.org/2007_overview_recent_supercomputers`, 11 May 2011.

[16] Gregory P. Pfister. *An Introduction to the InfiniBand^{TM} Architecture*, pages 617–632. John Wiley & Sons, Inc., 2001.

[17] Infiniband$^{\text{TM}}$trade association. `http://www.infinibandta.org/`, February 2011.

[18] Openfabrics alliance website. `http://www.openfabrics.org/`, 28 April 2011.

[19] Mellanox Technologies. *OpenSM User's Manual*, rev 1.20 edition, 2005.

[20] Robert Spence. *Information Visualization - Design for Interaction*. Pearson Education Limited, 2nd edition, 2007.

[21] C. Ware. *Information visualization: perception for design.* Morgan Kaufmann Publishers, 2004.

[22] C.J. Minard. *Carte figurative des pertes successives en hommes de l'Armée Française dans la campagne de Russie 1812-1813.* Graphics Press. Public domain.

[23] J. Bertin. *Graphics and Graphic Information-processing.* Walter de Gruyter & Co, 1981.

[24] W.S. Cleveland and R. McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79(387):531–554, 1984.

[25] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics (TOG)*, 5(2):110–141, 1986.

[26] B.B. Bederson, J.D. Hollan, K. Perlin, J. Meyer, D. Bacon, and G. Furnas. Pad++: A zoomable graphical sketchpad for exploring alternate interface physics. *Journal of Visual Languages and Computing*, 7(1):3–32, 1996.

[27] C. Ahlberg, C. Williamson, and B. Shneiderman. Dynamic queries for information exploration: An implementation and evaluation. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 619–626. ACM, 1992.

[28] H. Purchase, R. Cohen, and M. James. Validating graph drawing aesthetics. In *Graph Drawing*, pages 435–446. Springer, 1996.

[29] Chris Walshaw. A multilevel algorithm for force-directed graph-drawing. *Journal of Graph Algorithms and Applications*, 7(3):253–285, 2003.

[30] G. Di Battista, P. Eades, R. Tamassia, and I.G. Tollis. *Graph drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, 1998.

[31] Ivan Herman, Guy Melancon, and M. Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, 6(1):24–43, 2000.

[32] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[33] Reinhard Diestel. *Graph Theory*, volume Graduate Texts in Mathematics, Volume 173. Springer Verlag, Heidelberg, 2nd edition, 2010.

[34] G. Di Battista, P. Eades, R. Tamassia, and I.G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry-Theory and Application*, 4(5):235–282, 1994.

[35] Robert Tamassia. Handbook of graph drawing and visualization. `http://www.cs.brown.edu/~rt/gdhandbook/`, 12 April 2011.

[36] Arne Frick, Andreas Ludwig, and Heiko Mehldau. A fast adaptive layout algorithm for undirected graphs. In *Proceedings of the DIMACS International Workshop on Graph Drawing*, GD '94, pages 388–403, London, UK, 1995. Springer-Verlag.

[37] Stephen G. Kobourov. Handbook of graph drawing and visualization - force directed drawing algorithms. `http://www.cs.brown.edu/~rt/gdhandbook/chapters/force-directed.pdf`, 12 April 2011.

[38] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[39] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[40] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.

[41] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions on Graphics*, 15(4):301–331, 1996.

[42] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[43] Nam : Network animator. `http://www.isi.edu/nsnam/nam/`, June 2010.

[44] Stuart Kurkowski, Tracy Camp, and Michael Colagrosso. A visualization and animation tool for ns-2 wireless simulations: inspect. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 503–506, 2004.

[45] Björn Scheuermann, Holger Füßler, Matthias Transier, Marcel Busse, Martin Mauve, and Wolfgang Effelsberg. Huginn: A 3d visualizer for wireless ns-2 traces. In *Proceedings of the 8th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*, pages 134–150, 2005.

[46] The [ns] manual (formerly notes and documentation). `http://www.isi.edu/nsnam/ns/doc/index.html`, June 2010.

[47] Toilers- colorado school of mines- inspect. `http://toilers.mines.edu/Public/Code/Nsinspect.html`, January 2011.

[48] D. Schreiner, J. Neider, T. Davis, and M. Woo. *OpenGL®Programming Guide: The Official Guide to Learning OpenGL®, Version 2.1.* Addison-Wesley Professional, 6th edition, 2007.

[49] S. Kurkowski, T. Camp, and M. Colagrosso. A visualization and analysis tool for wireless simulations: inspect. Technical report, Technical Report MCS 06-01, Colorado School of Mines, 2006.

[50] S. Kurkowski, T. Camp, and M. Colagrosso. A visualization and analysis tool for wireless simulations: iNSpect. *ACM's Mobile Computing and Communications Review*, 2008.

[51] Huginn - visualizing network simulations in 3d. `http://pi4.informatik.uni-mannheim.de/pi4.data/content/projects/huginn/`, January 2011.

[52] Omnet++ manual. `http://www.omnetpp.org/doc/omnetpp41/manual/usman.html`, February 2011.

[53] Harald Meyer, Thomas Odaker, and Karin Anna Hummel. Omvis: a 3d network protocol visualization tool for omnet++. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10, pages 22:1–22:4, ICST, Brussels, Belgium, Belgium, 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

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

[55] D. Grune and C.J.H. Jacobs. *Parsing techniques: a practical guide*. Springer-Verlag New York Inc, 2nd edition, 2008.

[56] J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J.H. Wegstein, et al. Revised report on the algorithm language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963.

[57] I. Herman, M. Delest, and G. Melancon. Tree visualisation and navigation clues for information visualisation. In *Computer Graphics Forum*, volume 17, pages 153–165. Wiley Online Library, 1998.

[58] Frank van Ham. *Interactive Visualization of Large Graphs*. PhD thesis, Technische Universiteit Eindhoven, 2005. ISBN 90-386-0704-0.

[59] S. K. Feiner J. D. Foley, A. van Dam and J. F. Hughes. *Computer Graphics, Principles and Practice. Second edition in C*. Addison-Wesley Publishing Company, 1996.

[60] T. Akenine-Moller, E. Haines, and N. Hoffman. *Real-time rendering*. AK Peters, 2008.

[61] Ubigraph: Free dynamic graph visualization software. `http://ubietylab.net/ubigraph/`, April 2011.

[62] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman, 3rd edition, 1997.

[63] Devmaster.net - 3d game and graphics engines database. `http://www.devmaster.net/engines/`, February 2011.

[64] B. Karlsson. *Beyond the C++ standard library*. Addison-Wesley Professional, 2005.

[65] flex: The fast lexical analyzer. `http://flex.sourceforge.net/`, 04 May 2011.

[66] C. Donnelly and R. Stallman. *Bison: the YACC-compatible parser generator*. Free Software Foundation, 1995.

[67] conversion - c++ convert hex string to signed integer - stack overflow. `http://stackoverflow.com/questions/1070497/c-convert-hex-string-to-signed-integer`, 05 May 2011.

[68] J. Siek, L.Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[69] Lemon website. `http://lemon.cs.elte.hu/trac/lemon`, 18 April 2011.

[70] Introduction to lemon (presentation). `http://lemon.cs.elte.hu/pub/doc/lemon-intro-presentation.pdf`, 05 May 2011.

[71] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[72] Ogre - open source 3d graphics engine. `http://www.ogre3d.org`, 18 April 2011.

[73] Vtk - the visualization toolkit. `http://www.vtk.org`, 18 April 2011.

[74] Ogre: Api reference start page - ogre documentation. `http://www.ogre3d.org/docs/api/html/`, 18 April 2011.

[75] Ogre wiki - support and community documentation for ogre3d. `http://www.ogre3d.org/tikiwiki/`, 01 May 2011.

[76] G. Junker. *Pro OGRE 3D programming*. Apress, 2006.

[77] wxwidgets documentation. `http://docs.wxwidgets.org/stable/`, 18 April 2011.

[78] K. Henriksen, J. Sporring, and K. Hornbæk. Virtual trackballs revisited. *Visualization and Computer Graphics, IEEE Transactions on*, 10(2):206–216, 2004.

[79] wxwidgets website. `http://www.wxwidgets.org/`, 18 April 2011.

[80] Crazy eddie's gui system for games. `http://www.cegui.org.uk`, 18 April 2011.

[81] Qt - cross-platform application and ui framework. `http://qt.nokia.com/`, 18 April 2011.

[82] J. Smart, K. Hock, and S. Csomor. *Cross-Platform GUI Programming with wxWidgets*. Prentice Hall PTR, 2006.

[83] Qt online reference documentation. `http://doc.qt.nokia.com/`, 05 May 2011.

[84] Project wxformbuilder: The opensource wxwidgets designer, gui builder, and rad tool. http://wxformbuilder.org/, 19 April 2011.

[85] Ogre meshy at sourceforge.net. http://sourceforge.net/projects/ogremeshy/, 02 May 2011.

[86] wxogre website. http://owlet.sourceforge.net/wxogre/, 02 May 2011.

[87] Ogre forums - ogre/wxwidgets performance. http://www.ogre3d.org/forums/viewtopic.php?f=1&t=50502, 18 April 2011.

[88] Titan - usit/suf/vd/hpc compute cluster website. https://wiki.uio.no/usit/suf/vd/hpc/index.php/Titan, 05 May 2011.

[89] J. Nielsen. *Usability engineering*. Morgan Kaufmann, 1993.