Analysis of source code metrics from ns-2 and ns-3 network simulators

Juan Luis Font, Pablo Iñigo, Manuel Domínguez, José Luis Sevillano *, Claudio Amaya

Department of Computer Technology and Architecture, University of Seville, Seville, Spain

A B S T R A C T

Ns-2 and its successor ns-3 are discrete-event simulators which are closely related to each other as they share common background, concepts and similar aims. Ns-3 is still under development, but it offers some interesting characteristics for developers while ns-2 still has a large user base. While other studies have compared different network simulators, focusing on performance measurements, in this paper we adopted a different approach by focusing on technical characteristics and using software metrics to obtain useful conclusions. We chose ns-2 and ns-3 for our case study because of the popularity of the former in research and the increasing use of the latter. This reflects the current situation where ns-3 has emerged as a viable alternative to ns-2 due to its features and design. The paper assesses the current state of both projects and their respective evolution supported by the measurements obtained from a broad set of software metrics. By considering other qualitative characteristics we obtained a summary of technical features of both simulators including, architectural design, software dependencies or documentation policies.

Keywords:
Discrete-event simulation
ns-2
ns-3
Software metrics

1. Introduction

Simulation is a key tool in networking research due to its inherent advantages over testing with real hardware [1]. Matters such as budget and cost of time make simulation attractive for research, while flexibility, scalability and virtually no cost expansion are also valuable advantages over other testing methods [2]. Network simulators allow us to implement and study different network entities in a simulated environment, providing a high degree of flexibility to test new protocols, technologies, conceptual models and topologies.

Ns-3 [3] has been conceived as ns-2's successor: its developers have tried to solve or mitigate many of ns-2's well-known drawbacks as well as apply new concepts, such as validation and software engineering techniques, to produce a more reliable simulation tool to support academic and industrial research [4].

Other technical papers have already compared ns-2 and ns-3 [5], along with other network simulators, evaluating their core performance and scheduling capabilities but not focusing on any specific simulator features to make the comparison of very different pieces of software as fair as possible. Instead this study has focused on two specific simulators, ns-2 and ns-3, which are quite close in terms of structure, aims and user base. These facts allowed us to focus on technical aspects that are not directly comparable in other simulation packages due to their quite different characteristics and nature. In other words, adding other simulator packages to this study would introduce too different elements that are not susceptible to be compared in the same terms than with ns-2 and ns-3. One good reference that has good review of other simulation packages that the interested reader may want to look at is Obaidat and Papadimitriou [6].

* Corresponding author. Address: ETS Ingenieriá Informática. Avda. Reina Mercedes s/n. 41012, Seville, Spain. Tel.: +34 954556142.
E-mail addresses: juanlu@atc.us.es (J.L. Font), pabloinigo@atc.us.es (P. Iñigo), mdominguez@atc.us.es (M. Domínguez), sevi@atc.us.es (J.L. Sevillano), camaya@atc.us.es (C. Amaya).

This paper is an extension of a previous article [7], which made a technical comparison of topics such as software architecture, design and suitability of use for developers interested in network simulation. The former article, laid the groundwork for this paper, which picks up from the technical comparison deployed in the previous work, extending the analysis and interpretation of several software metrics extracted from both network simulators. It provides a more thorough analysis of how they have evolved by analyzing their respective source code releases. Subjects such as source code hierarchy, generation of binaries, portability or structuring and documenting policies are also discussed in this paper, with the latest source code metric analysis being the vehicle for supporting many of the conclusions extracted from each of the above areas.

This work attempts to provide a comprehensive analysis to evaluate the suitability of the use of ns-3 as an alternative to ns-2 by highlighting technical features and providing measurements that can be extrapolated as a proof of the maturity of the project, while not encouraging the use of any of the two simulators which is a decision by the end user.

The comparison of both network simulators is divided into different sections which are focused on different topics related to the properties of the architecture and source code of both projects. The present comparison starts in the following section, Section 2 which deploys a study of the system integration properties of both software tools taking into account subjects such as their respective distribution policies, source code releases, libraries, third-party software dependencies and the source code building process. The following section, Section 3, focuses on project design and architectural issues, looking at ns-2 dual-language design, general source code organization of both projects, documentation and portability subjects. Section 4 gathers and summarizes the information derived from the study of several procedural and object oriented source code metrics, making an interpretation of the figures obtained in order to elaborate a study of the evolution of both projects over the course of their respective releases. The papers ends by bringing together the different conclusions followed by Acknowledgments and then Bibliographical references.

2. System integration

The Ns simulators family has its roots in Unix-like environments, having been developed using typical Unix languages and tools. Nowadays, GNU/Linux is one of the main development platforms for both ns-2 and ns-3 [8]. Although they have been ported to other operating systems such as Windows/Cygwin [9], *BSD or Mac OSX [10], it is easier to install, update and maintain instances of both simulators by working under GNU/Linux environments.

2.1. Distribution

Both ns-2 and ns-3 were conceived to be distributed primarily in source code form. As users have access to source code, they can modify and extend their features and optimize their binaries as required. Source code can be fetched from their respective project repositories. Ns-2 relies on CVS [11], a classical revision control system, to manage its source code, while ns-3 uses Mercurial[12], an emerging distributed revision control tool.

Ns-2 has several software dependencies, some of them can be obtained through the software repositories of main GNU/Linux distributions, but others have to be fetched and installed manually due to their absence in mainstream software repositories [13]. Depending on very specific, and sometimes, obsolete library versions represents an important drawback when installing ns-2. This problem is mitigated by project maintainers through packaging the whole set of dependencies in one single installation package, aka *all-in-one*. This package contains both ns-2 and libraries source code, configuration files and the tools needed to compile the whole project. This installation alternative is preferable to fetching source code from ns-2 CVS repository and trying to solve problems with library paths and compilation errors manually.

Like ns-2, ns-3 offers an *all-in-one* installation package [14] which only contains ns-3 source code ready to compile. During the compilation process, the ns-3 compilation scripts will inform about the missing software dependencies that can be easily obtained from different sources.

2.2. Specific software dependencies

GNU toolchain is the first requirement for building ns-2 or ns-3. Currently there are no reported issues related to recent gcc versions. Apart from the toolchain, ns-2 and ns-3 have several library dependencies. The main ones required for ns-2 are:

- Tcl/Tk: Tcl scripting language and Tk, a Tcl extension that provides graphical user interface library. The current Tcl stable version is 8.5.8 (2009-11-16) [15], although ns-2 developers state that the simulator has only been tested against 8.4.14.
- OTcl: a Tcl object-oriented extension [16], it is used as ns-2 scripting language for writing simulations. The last release dates from 2007-03-10 and for the last 5 years their developers have only released minor versions (1.8–1.13). It is not found as a standard software package in most GNU/Linux software repositories due to its limited diffusion and use.
- TclCL: is Tcl/C++ interface that allows the creation of OTcl wrappers for C++ code. The development of TclCL is closely related to OTcl. Like OTcl, TclCL is not a widespread tool and is rarely found in software repositories of the main GNU/Linux distributions.

Ns-2 can be found as a regular software package in some GNU/Linux distributions, such as Debian, which includes ns-2 in its testing and unstable branches [17]. The maintenance of this piece of software in the repositories depends on particular maintainers of the respective projects that package and distribute ns-2.

Ns-3 also requires external libraries, but they are quite standard pieces of software and are usually available as part of the main GNU/Linux distributions. Only development libraries such as libxml and python are mandatory. The remaining dependencies will be installed optionally to enable certain specific features. Software such as valgrind, sqlite or GNU Scientific Library are examples of these optional dependencies. They are all easy to find in the repositories of mainstream distributions such as Debian, Ubuntu or Fedora [8]. Packaging tools make software installation much neater and easier by automatically resolving dependencies and configuring library paths. In addition, there are no major issues related to using brand new library versions, which are soon tested against current ns-3 code. Ns-3 can also be found as a standard software package ready to install in some distro repositories such as Debian, which includes ns-3 as a package in its testing and unstable branches [18].

In conclusion, ns-3 has a better system integration with its host operating systems, especially GNU/Linux. There is low coupling between ns-3 itself and its required libraries and tools, so users have a higher grade of flexibility for solving software dependencies independently of their specific host operating system. This advantage is almost mandatory for a software and is still under active development that is distributed throughout the world. In contrast, Ns-2 shows signs of aging and its development seems to be stuck, the project itself cannot use too many resources on testing and updating external software dependencies so ns-2 ultimately relies on mostly obsolete versions which is a major drawback for installation and maintenance processes.

2.3. Building the source code

Ns-2 uses a classic Unix configuration and building process based on the `make` building tool. To add new modules to ns-2 or change current configuration, users themselves must edit the project's main Makefile [19], and any change on its configuration means rebuilding the whole source code. Although a certain amount of time is saved by using scripting languages for writing simulations, any change related to ns-2 core and its C++ source code needs a whole project compilation. The core and modules building process generates a monolithic executable called `ns` which contains the simulator functionalities as well as all the current models distributed with ns-2. Simulations are just OTcl scripts, but they need `ns` binary to run. Scripts are passed as parameters to `ns`, which acts as a script interpreter. The building process takes an average compilation time of 1 m 43 s on a computer equipped with an AMD Quad Core 2'6 GHz processor, 4 GB of RAM, using Debian 5.0.3 for amd64 architecture and gcc 4.3.4.

Ns-3 supports Python scripting, although writing simulations in this language are not mandatory and Python support and its wrappers can be optionally enabled. Ns-3 also relies on Python for configuring and compiling its own source code [20]. Instead of using `make` tool as ns-2 does, ns-3 has adopted `waf` as a build automation tool [21], which has Python as its single external dependency. Like Python language itself, `waf` is portable. It only allows compilation of modified source code files, ignoring the untouched ones. Rebuilding a relatively simple class takes only between 1 and 2 s when using the same AMD Quad Core, 4 GB RAM based machine running Debian and gcc 4.3.4. After compiling the whole project, the user gets a monolithic shared library called `libns3.so` and a single executable for each simulation written in C++ language. These binaries have to be dynamically linked with `libns3.so` at runtime. Therefore, the ns-3 binary core is more a shared library than a traditional executable such as ns-2. `Libns3.so` contains the simulation routines as well as all the ns-3 standard modules.

3. Project design and source code properties

The general architecture, code structuration and other developer-related topics are discussed in this section. These issues are approached qualitatively, with quantitative metrics being left for the next section.

3.1. Dual-language architecture

The widespread use of Tcl and its language extensions in ns-2 responds to a past context in which compiled languages such as C++ were relatively high time-consuming for the hardware of the time, so ns-2 developers chose a dual-language architecture. They used C++ for core elements and models, which were supposed to be more stable and static pieces of software, compiled once and run many times. They also chose a scripting language such as Tcl and its object-oriented extension, OTcl, for writing simulations that would run over ns-2. Using non-compiled language allowed users to write and run simulations without suffering from long compilation times.

The dual-language design defines the way ns-2 users and developers write code. Simulation routines and models are implemented in C++, a compiled language, in order to benefit from optimization and speed up. Simulations themselves are only OTcl scripts that invoke the functionalities coded in C++ thanks to software wrappers [16]. Despite reducing compilation times, the dual-language nature adds extra complexity to ns-2 use and development. It can be difficult for new developers to identify which parts have to be coded in C++ and which ones in OTcl language when coding a new model.

Today's hardware is powerful enough to deal with compiled languages like C++, consuming negligible compilation times in most cases, so the advantages derived from using a scripting languages such as OTcl are steadily decreasing [22].

Ns-3 has simplified its design choosing C++ as the sole development language for its models', simulations can also be coded in C++. Ns-3 has not totally abandoned the use of scripting languages for coding simulations, but it has changed the way they are used in the project [23]. Compilation times are no longer an issue, so the advantages from using scripting languages stem from other technical reasons such as portability, productivity or easier syntax. Ns-3 users have the option of choosing Python as the language for writing simulations thanks to Python bindings. Shifting from OTcl to Python provides a scripting language with a larger user base, more active development and friendlier syntax.

3.2. Architecture and code organization

Ns-2 defines a very basic architecture, only distinguishing the core of the simulator and network models. This fact has become a drawback as ns-2 has evolved and new modules and features have been added. Nowadays some modules show a high degree of coupling and dependencies are difficult to follow [24].

Ns-2 differentiates between the network topology and the agents that run over it and generate or consume data traffic [25,2]. The topology is formed by generic nodes and the links between them. Furthermore, these links can have special objects associated that define link behavior [26], channel characteristics and other parameters. Agents are the network applications themselves that run over the nodes and transmit or receive data information through the network. Model developers are responsible for ensuring that the agent layer will be able to exchange data packets with the network topology layer, so they have to design their own data packets if necessary and register them as new types so that ns-2 can handle them. Packet registration means users have to change some ns-2 core source code files and rebuild the whole simulator [27], which is quite an invasive process.

Apart from the differentiation between topology and agents, ns-2 programmers have a high degree of freedom to define and code their models, so, depending on the desired abstraction level, they can write from very simple and abstract models to relatively complex and accurate ones. Thus, simulation results must be analyzed carefully, depending on the level of abstraction applied, some simulation results can diverge considerably from real-world ones [4].

The ns-2 relaxed hierarchical organization is reflected in its source code structuring. Early releases did not make any distinction at all between modules, and all the source code files were placed at the same level on a common folder. From the release of 2.26 onwards, maintainers have attempted to keep a proper source code structure [28].

On the other hand, ns-3 emphasizes source code hierarchical structuring by defining several basic network entities present in every single simulation. Specific models are a refinement of these generic entities [29].

- *Node*: represents the basic computing device. It gathers the network functionality and interacts with other nodes through the communication channel. Nodes require net devices to be able to use the physical channel.
- *Application*: sets up on top of the nodes, playing the role of packet consumer or packet generator.
- *Channel*: provides the medium for interconnecting nodes and allowing data traffic.
- *Net Device:* abstracts the network hardware that makes communication possible through the channel.
- *Topology Helper:* auxiliary class that makes the generation of complex topologies easier by automating the network elements' creation, configuration and interconnection.

Ns-3 gathers all source code files of both simulator core and models in the `src` folder, separating them from the remaining auxiliary tools, building scripts, documentation and examples [30]. The `src` folder is also subdivided, to cater for the above classification of abstract entities. Thus, all the models related to network devices can be found in `src/devices`. This hierarchical organization makes it easy for developers to navigate through the simulator code. Following the above scheme, ns-3 programmers are encouraged to define several entities in their abstract model to fit their code in the global simulator structure, ensuring a common model skeleton that makes the overall code more flexible and adaptable, and easier to refine and reuse in the future.

Since the `src` folder has been conceived just for containing the standard ns-3 modules, which are maintained, supported and documented by the ns-3 project itself, new third-party modules should be created and built into the `ns-3_home/scratch`. The `waf` tool scans the `scratch` folder and generates an executable for each defined simulation.

3.3. Simulation portability

Ns-2 simulations are initially portable since they are simple OTcl scripts. There is no problem in launching an ns-2 simulation in a different ns-2 instance, regardless of the target architecture of the `ns` binary that will run the script. This is true provided that ns-2 scripts only use ns-2 standard features included in the official source code release. If the script uses custom models developed by third-parties that are not official parts of the ns-2 project, it will need a custom `ns` executable, generated from the standard ns-2 source code plus the new model code [31]. Thus, the ns-2 design attaches a simulation that uses custom models for the specific ns-2 instance containing them, losing any kind of advantage concerning portability due to the need for a custom `ns` interpreter.

Ns-3 design overcomes the above drawback by providing a common `libns3.so` library that must be the same for all the ns-3 instances belonging to the same release. If developers use the `scratch` folder to place new models and simulations, they will obtain standard binaries that need `libns3.so` shared library in order to use ns-3 standard features, and they will include the custom models as part of their own binary code. These executables are only constrained by their target processor architecture (i386, amd64, ppc, etc.).

3.4. Source code documentation

Ns-2 does not establish any criterion related to documentation. The project maintains a web page with its main documentation dispersed in several sections, mixing legacy documents, tutorials and third-party manuals. This model may suffer from problems such as obsolescence and lack of maintenance.

Documenting ns-2 can be viewed as a double and non-automated task. There is no official policy concerning ns-2 source code documentation, so developers apply their own criteria in this matter. Thus, the use of tools, which can extract information from the comments of the source and generate proper documentation in other formats, does not ensure the generation of a comprehensive and complete source code documentation. The impossibility of generating developer-oriented documentation automatically means that official documentation has to be written manually. This leads to a situation where the documentation contained in the comments and the official documentation are independent text that require manual maintenance and synchronization.

Ns-3 integrates documentation in source code files by using Doxygen [32], an open-source multi-language documentation generator, which parses and extracts ns-3 documentation directly from source code. It allows defining some format aspects of the documentation such as defining parameters, return values, and links to others pieces of documentation. In addition, documentation can be generated in several different formats such as pdf, HTML, latex or XML. Thus, generating ns-3 documentation is as simple as compiling its source code, and at the same time it removes the need to maintain two different documentation sources. This way, commenting code and writing official documentation are a single task and users can get this information in the form that best suits their needs [33].

Stats show ns-2 has a higher ratio of lines of code per comment line [7]. Documentation policies have provided a more exhaustive and up-to-date documentation of the ns-3 project, which is better documented both qualitatively and quantitatively.

4. Quantitative analysis of source code

C++ source code files from several ns-2 and ns-3 previous and current releases have been analyzed to extract basic software metrics (ns-2: from 2.0 to 2.34, 13+ years, ns-3: from 3.0 to 3.8, 3+ years) [28,34]. Ns-3 is shorter lived than ns-2, but it has benefited from ns-2 ideas and experience, so it cannot be considered as having been developed from scratch. On the other hand, ns-2 has gone through different stages during its long existence and it can now be considered to be nearing the end of its life cycle, with bug fixing and maintenance releases being its main priorities.

As mentioned earlier this paper extends the work of a previous related article [7], going deeper into source code study through calculation and interpretation of software metrics. The brief analysis of metrics shown in the previous paper has been revised, updated and recalculated to provide a more detailed and accurate statistical analysis and more in-depth interpretations.

4.1. Source code and metrics analysis tool

Both network simulators use C++ as programming language for developing their core components and models [2], but they also rely on other scripting languages to develop auxiliary code to make integration, maintenance or simulation development easier. One example of this is ns-2's use of Tcl/OTcl; and Python is used as an optional simulation development language in the context of ns-3.

This comparative study has focused on the analysis and review of the respective core of each network simulator. These elements are subject to architectural decisions and they deal with many of the simulation work load during execution time. As stated above, the core elements are written in C++ in both projects, so the auxiliary code developed using scripting programming languages have not been taken into account for the metric analysis.

A set of well-know and widespread used software metrics have been chosen to perform the measurements on the source code of both projects, since the purpose of the use of these metrics is providing a quantitative analysis of the evolution and current technical state of both tools. Newer and more novel software metrics have not been taken into account since more conventional ones have already provided the results needed to support the present work.

Since this study focuses solely on C++ code, the software metric tool cccc [35] (C and C++ Code Counter) has been chosen to perform the source code analysis. Its features, availability and condition as free software under the GPL as well as its specialization in C and C++ languages make it a suitable tool for the purposes of this paper.

Ns-3 incorporates third party C code, related with Linux kernel headers [36], which was susceptible of including high rates of noise in the source code analysis. These source code files have not been considered as part of

ns-3 project itself, so they have been left out of the analysis process to obtain a more representative analysis and comparison.

The data mining process has been automated using Unix shell scripting programs, which have allowed us to perform source code analysis of each release and parse and summarize all the results in a simpler set of plain text files.

4.2. Source code releases and reference dates

All the major source code releases of both network simulators have been obtained from their respective repositories. In the case of ns-2, having been under development for so long, and changes in repository, infrastructure and organization, there is a gap in the list of ns-2 releases studied in this paper. This gap spans the time from the first major releases, ca. 1997, to the next sample, the ns-2.1ba release, dated in 2000 [37,38]. Despite this minor drawback, we have focused on the latest ns-2 samples, rather than the early releases. We took this decision to make it possible to analyze comparable network simulator versions in terms of maturity and functionality.

4.3. Roadmap and releases

Several ns-3 and ns-2 releases have been analyzed to evaluate their respective evolution by extracting some basic software metrics. The list of analysed releases can be found in Tables 1 and 2. Both current and previous releases have been fetched from their respective official source code repositories and archives [39,40].

Ns-2 started as a variant of the REAL network simulator in 1989. Today, its changelogs [37] reflect bug fixes and general code maintenance, without an official roadmap. Releases 2.31, 2.32 and 2.33 were not available on the ns-2 archive [38].

The Ns-3 project started in 2006 and initially was initially planned to take 4 years. It is still currently under active development, having several full-time hired maintainers [41]. Since it depends on contributed modules, it is difficult to define an exact project roadmap [42].

4.4. Procedural metrics

Although the overall results cannot be compared directly due to the different amount of features that ns-2 and ns-3 implement, it is worth analyzing and comparing averages to obtain a more realistic and fair code comparison. The following procedural software metrics have been extracted:

Table 1
Ns-2 versions by release date [37].

	Version	Release date
1	ns-2.0	May, 1997
2	ns-2.1b1	May, 2000
3	ns-2.1b7	October, 2000
4	ns-2.1b8	June, 2001
5	ns-2.1b9a	July, 2002
6	ns-2.26	February, 2003
7	ns-2.27	January, 2004
8	ns-2.28	February, 2005
9	ns-2.29	October, 2005
10	ns-2.30	September, 2006
11	ns-2.31	March, 2007
12	ns-2.32	September, 2007
13	ns-2.33	April, 2008
14	ns-2.34	June, 2009

Table 2
Ns-3 versions by release date [43].

	Version	Release date
1	ns-3.0.1	March, 2007
2	ns-3.0.13	June, 2008
3	ns-3.1	July, 2008
4	ns-3.2.1	September, 2008
5	ns-3.3	December, 2008
6	ns-3.4	April, 2009
7	ns-3.5.1	September, 2009
8	ns-3.6	October, 2009
9	ns-3.7.1	March, 2010
10	ns-3.8	May, 2010

- NOM: Number of modules. Number of non-trivial modules identified by the analyzer.
- LOC: Lines of code. Number of non-blank, non-comment lines of source code.
- COM: Lines of comments. Number of lines of comment identified by the analyzer
- MVG: McCabe's cyclomatic complexity. A measure of the decision complexity of the functions which make up the program. The strict definition of this measure is that it is the number of linearly independent routes through a directed acyclic graph which maps the flow of control of a subprogram.
- L_C: Lines of code per line of comment. Indicates density of comments with respect to textual size of program
- M_C: Cyclomatic complexity per line of comment: Indicates density of comments with respect to logical complexity of a program.

4.4.1. Number of modules: NOM

According to cccc documentation, a module corresponds to classes and C/C++ modules with member functions [44]. Regarding ns-2 and ns-3, both of their cores are written in C++ following the Object Oriented Paradigm (OOP) so each module identified by cccc during the parse process corresponds to a C++ class.

A moderate value for this metric may indicate scope for code reuse, however high values may be an indication of inappropriately high levels of design abstraction [44]. Due to the nature of the reality modeled in both network simulators, there is a correspondence between most modules and real world entities: channel, network interfaces, stacks, frame types, etc. [29]. Therefore excessively high levels of abstraction are not a major issue and they might only be observed in specific and isolated cases (for example, ns-3 uses auxiliary classes to implement different software patterns such as factory, singleton and so on), mostly associated with architectural and modular design, so their use is justified [45].

In contrast, this metric is closely related to the amount of features and growing complexity of each simulator project [46,37]. These features can be divided into two categories: network simulator internals (scheduler, system integration, software patterns) and network models for different networking technologies (support for different technologies such as Ethernet, Wifi, mobility models, propagation models and so on) [47]. There is a logical increase in the number of modules associated with the increase of features in each network simulator.

The graph covering the evolution of both simulators between releases shows important increment in the number of modules for ns-2 between the initial and final releases of the ns-2.1bX series (see Fig. 1.a). This can be explained by the addition of a substantial number of models during the development of the ns-2.1bX series. Starting with a bare-core, official support for several models was added during this period. On the other hand, ns-3 shows a steady evolution during the mid releases, boosting the size and complexity of the project in recent releases (see Fig. 1.b).

At present,the size of both projects in terms of complexity and size is comparable. A brief look at the list of features of both simulators reveals that ns-3 already covers and implements many of the fields where ns-2 has been used up until now. Ns-3 still lacks models for specific niches that are well supported by ns-2. In contrast, ns-3 has overcome ns-2 in some architectural aspects as well as utilities and auxiliary methods that allow different levels of log output, modularization, automation of the building process and so on. It is worth highlighting the widespread use of software patterns, which increases their software complexity in terms of auxiliary classes.

Judging by the stagnation that can be observed in ns-2 development, as well as the ascending evolution of its successor, it seems fair to assume that ns-3 will surpass ns-2 in terms of size, defined as the number of modules that make up both core and network element models. This overcome will happen as soon as ns-3 offers a similar set of models to ns-2, plus the

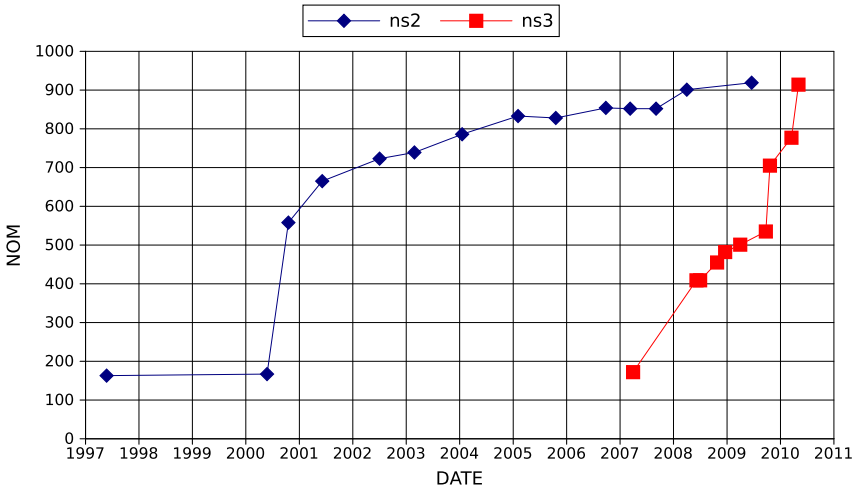

Fig. 1.a. Total number of modules over time.

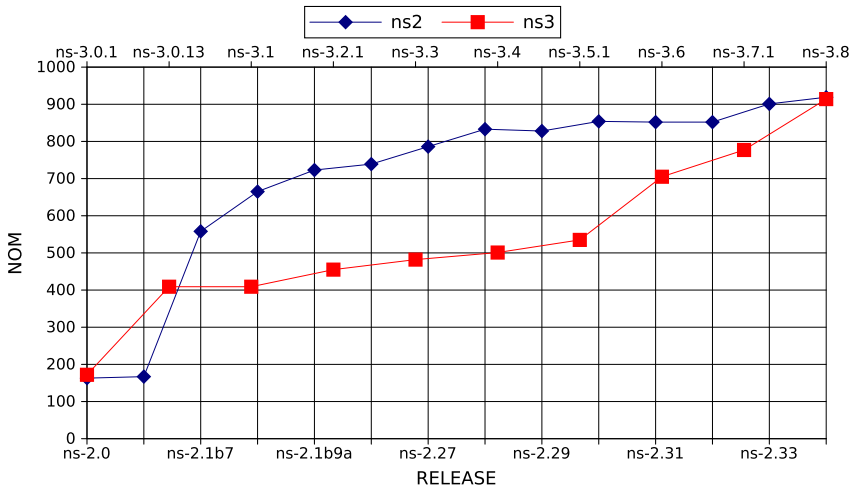

Fig. 1.b. Total number of modules vs. release.

centralized architecture that defines the relationship between core, current and future new models which means a larger number of modules than ns-2 in order to implement the core functionalities.

4.4.2. Lines of code: LOC

LOC is defined as the number of non-blank, non-comment lines of source code [44]. Figs. 2.a and 2.b show a similar size of both projects in terms of total lines of code, as well as a very similar evolution and increment between their respective releases.

The ns-2.1bX series represented a great boost in terms of complexity and size of the project as can be observed in both graphs. The ns-2 curve is less pronounced than the ns-3 one. The latter has recently overcome ns-2 default distribution in terms of raw size.

A second interpretation of the values obtained for the LOC metric can be made in conjunction with the above NOM metric. The result from dividing the total lines of code of each project into their number of modules gives an approximation of the average size of each class, which can be taken into account as one of the indicatives of complexity [50].

Calculations show the LOC/NOM relation remaining quite stable in both projects throughout their releases, with ns-3 having slightly bigger modules on average, although this does not necessarily mean more complex ones (see Figs. 3.a and 3.b).

4.4.3. McCabe's cyclomatic complexity: MVG

McCabe's cyclomatic complexity (MVG) gives a measure of the decision complexity of the functions which make up the program [44,48]. The strict definition of this measure is the number of linearly independent routes through a directed acyclic graph which maps the flow of control of a subprogram.

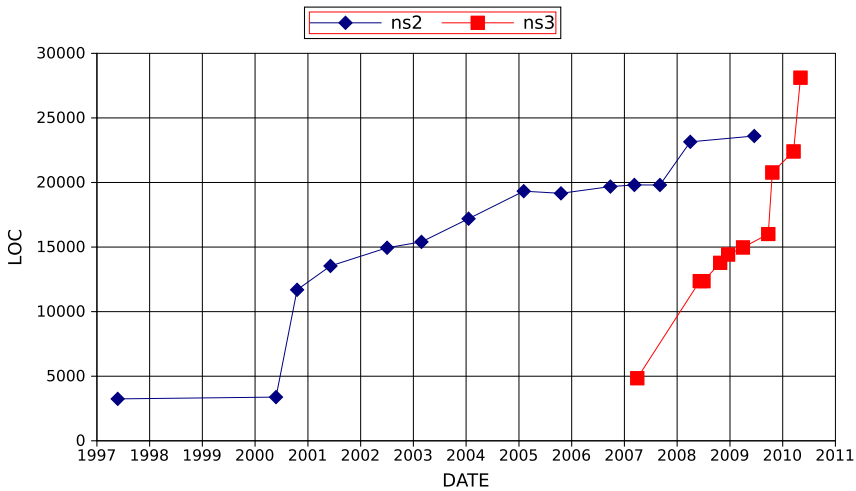

Fig. 2.a. Total number of lines of code over time.

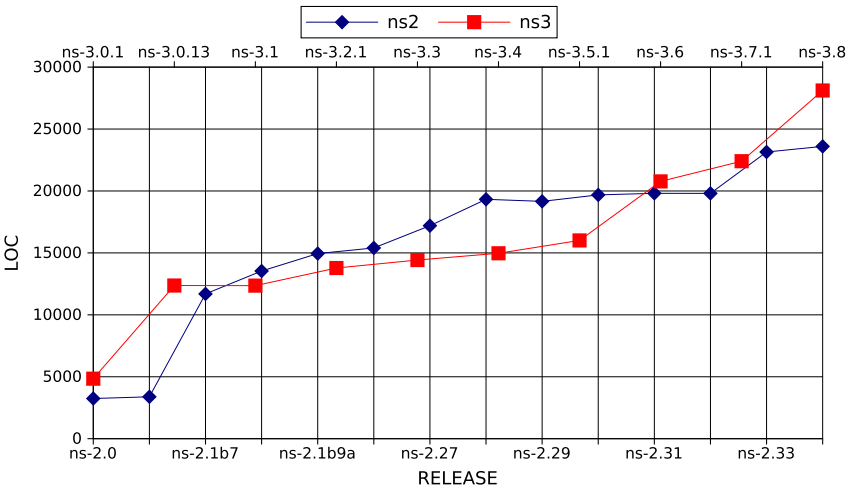

Fig. 2.b. Total number of lines of code vs. release.

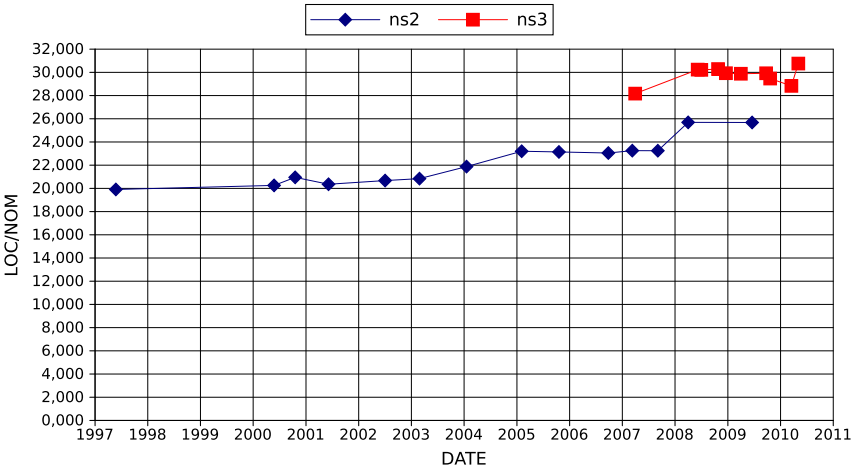

Fig. 3.a. Relation of lines of code per module over time.

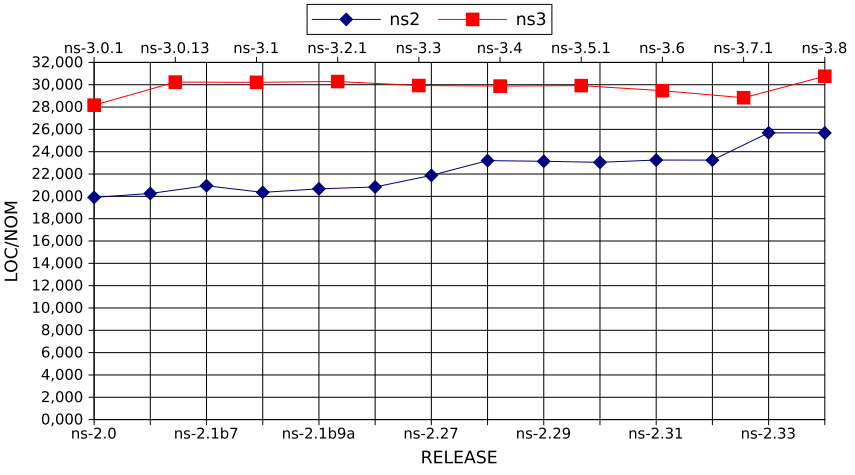

Fig. 3.b. Relation of lines of code per module vs. release.

MVG can be regarded as an indicator for *Cohesion* [49] in the source code, considering that a code with higher complexity has a lower degree of cohesion [50,51]. The possible correlation between the higher complexity and lower cohesion is based on the assumption that a module with more decision points probably implements more than one single and well-defined function.

The complexity measured by the MVG also correlates with errors contained in the source code. A number of studies have found a strong correlation between complexity and defects [52], while others have not found that correlation [53].

The figures obtained for both network simulators differ considerably in terms of cyclomatic complexity, Figs. 4.a and 4.b. While ns-3 shows a very stable and almost constant degree of complexity in spite of its constantly growing size, ns-2 reveals a clear increment in MVG values in different stages of its development, also having periods of stagnation.

One of the reasons that may explain these figures are the architectural policies adopted by ns-3 [29], that have allowed a sustainable and well scaled development while ns-2 has had to face different development challenges throughout its development cycle up to a point where it is difficult to accomplish new goals using its original architecture [22].

The reading that can be drawn from MVG and LOC metrics in conjunction is that the bigger ns-3 modules are not necessarily more complex than ns-2's, so the raw extra size does not translate into higher degrees of linearly independent routes through the source code.

4.4.4. Lines of comments: COM

The COM metric corresponds to the number of lines of comment identified by the analyzer [44]. The figures correspond to the whole number of lines of comment per project.

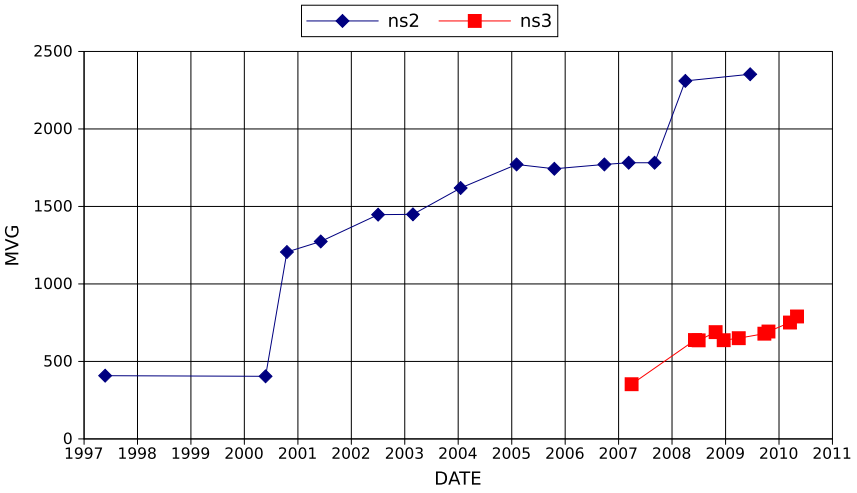

Fig. 4.a. McCabe's cyclomatic complexity over time.

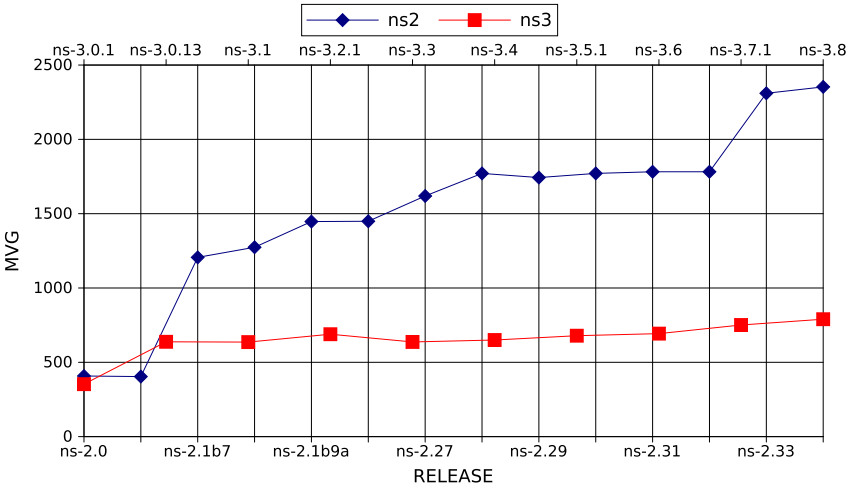

Fig. 4.b. McCabe's cyclomatic complexity vs. release.

As stated in the previous comparison, ns-3 overcomes ns-2 in terms of code documentation thanks to its policy concerning documentation for developers. Thus, ns-3 programmers are encouraged to document each part of their source code using the Doxygen syntax to automatically generate development documentation [54]. In contrast, ns-2 does not reinforce any specific policy related to documentation, so different approaches can be found in its source code, from modules documented to others that lack that kind of in-line documentation and requires third party sources to describe the behavior and technical details of the code.

This statement is clearly observed in the figures Figs. 5.a and 5.b. While both network simulators have been leveled for their respective early versions, recent ns-3 releases clearly overcome the latest ns-2 releases in total number of lines of documentation.

Comparing the increments of lines of code and comments in ns-3 suggests that the steady growth of the project is accompanied by a more detailed documentation of its elements. The lines of code per line of comments (L_C) metric reinforces this hypothesis (Figs. 6.a and 6.b). Ns-2 shows a relatively constant value for this relation between lines of code and lines of comment while ns-3 shows a smooth downward curve.

4.4.5. Cyclomatic complexity per line of comment: M_C

The M_C metric indicates density of comments with respect to logical complexity of the program [44]. In previous sections ns-3 has show better figures for both number of lines of comment and cyclomatic complexity. Thus, the M_C metric derived from the two previous metrics, Figs. 7.a and 7.b, reflects the same behavior, with ns-3 being the one with better

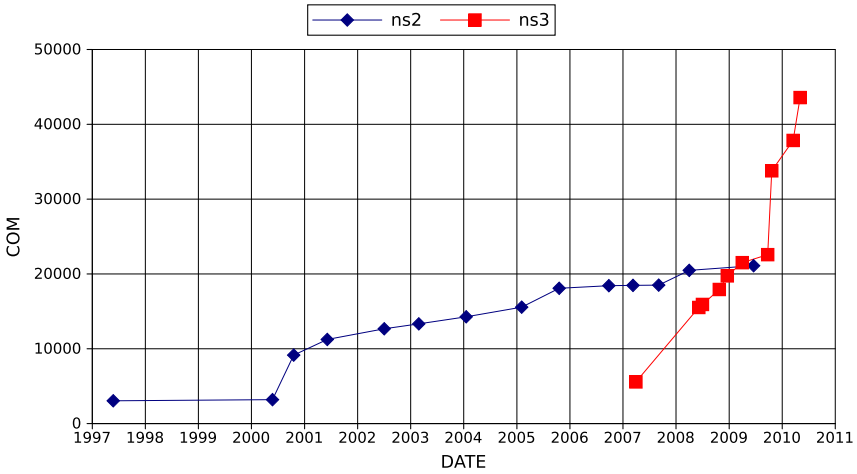

Fig. 5.a. Total number of lines of comments over time.

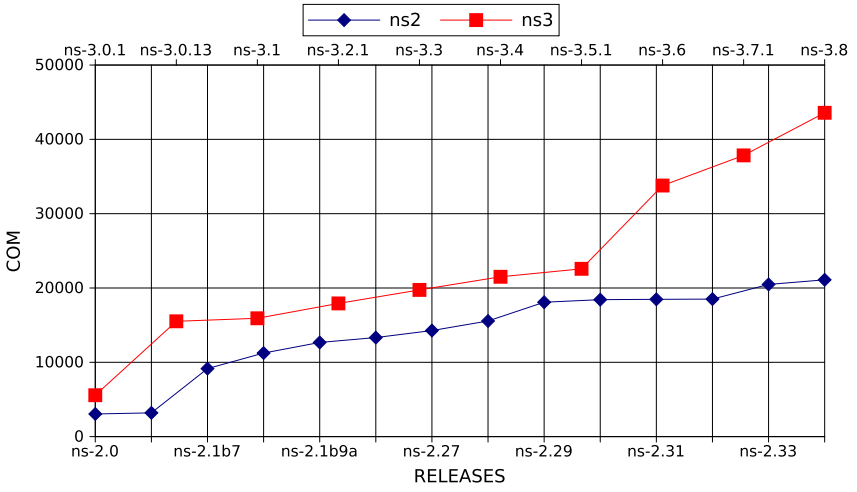

Fig. 5.b. Total number of lines of comments vs. release.

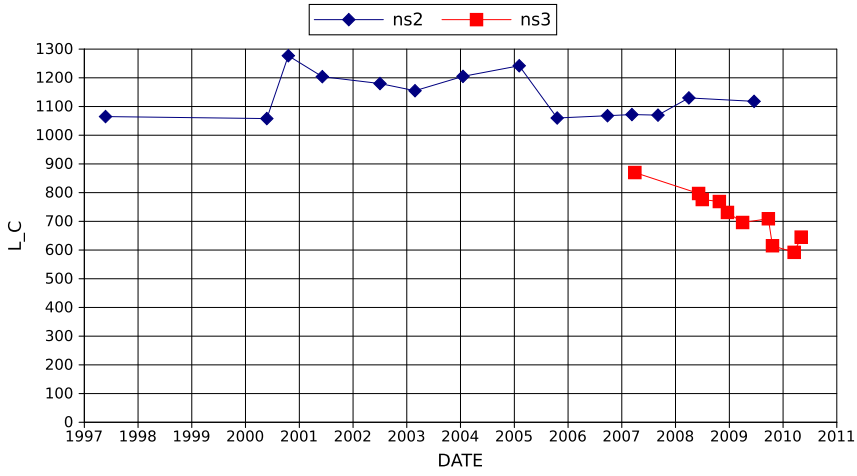

Fig. 6.a. Relation of lines of comments per line of code over time.

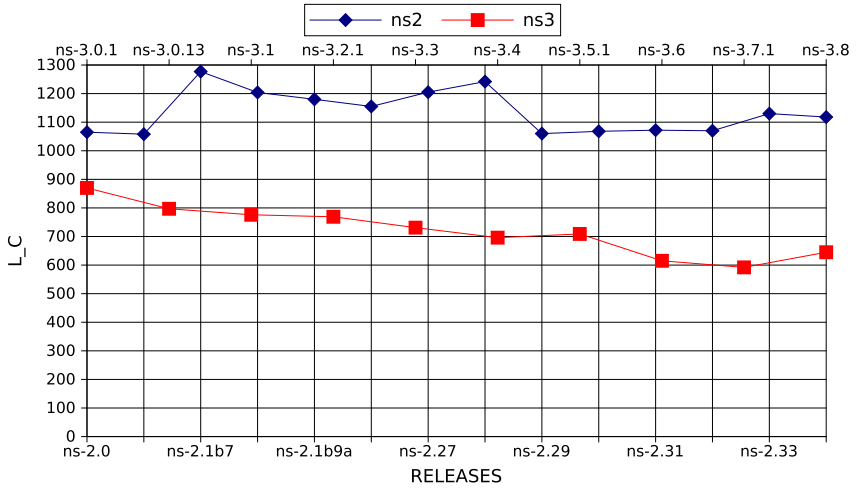

Fig. 6.b. Relation of lines of comments per line of code vs. release.

results, with lower figures for this metric being a good indicator which could provide the basis for a solid documentation linked to cyclomatic complexity.

4.5. Object Oriented Design Metrics

The procedural metrics analyzed in the above section measured characteristics from each single release, such as number of lines of code and number of modules. In this case, the basic element taken into account during the analysis was the software release.

In contrast, the Object Oriented Design Metrics [55] measure each single object that integrates the software release. Thus, for each simulator release there is a large set of objects, whose items vary from one release to another, depending on architectural redesigns, addition of new features or complementation of the existing ones.

While the procedural metrics were intended to give a brief description of both projects' size and evolution, the Object Oriented ones aim to show the current architectural status of both network simulators. Thus it is worth focusing on the most recent releases of each network simulator to gather key information about design, encapsulation and source code reusability.

Instead of showing the evolution over time of each Object Oriented Design metric for a large and changeable set of objects, we thought it more worthwhile to analyze the last release of each network simulator (ns-3.8 and ns-2.24 respectively) and perform a more detailed analysis of the considered Object Oriented Design metrics, attaching statistical descriptors such as mean, deviation, minimum, maximum, mode or median (Tables 3 and 4).

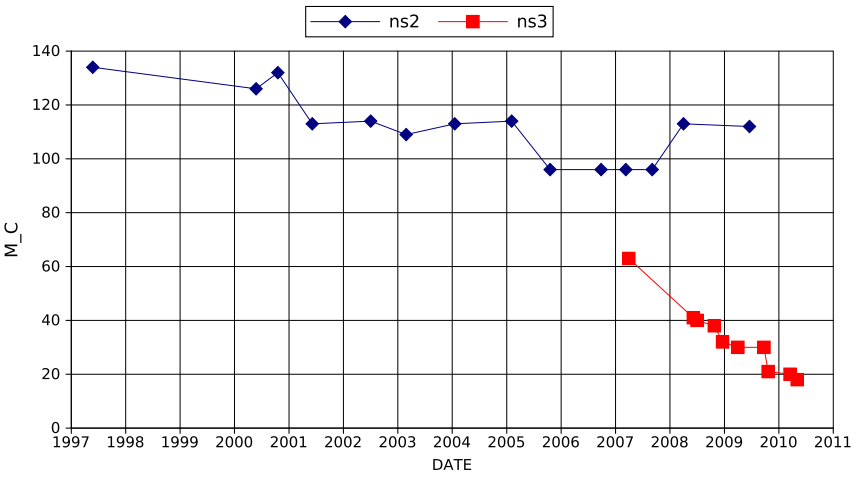

Fig. 7.a. Relation between lines of comment and cyclomatic complexity over time.

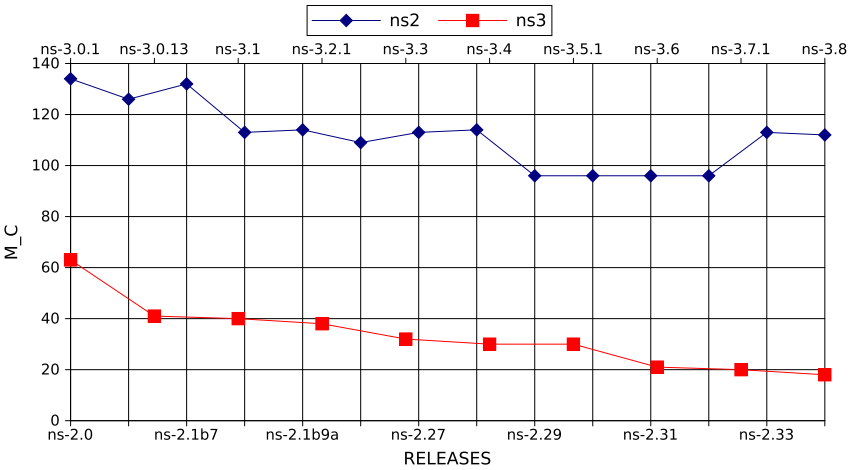

Fig. 7.b. Relation between lines of comment and cyclomatic complexity vs. release.

Table 3
Ns-2.34 Object Oriented Design stats.

	WMC1	WMCv	DIT	NOC	CBO
Mean	6.76	6.07	1.74	0.68	**5.56**
Deviation	17.58	12.34	1.65	4.45	**13.90**
Min	0	0	0	0	0
Max	465	300	7	97	255
Mode	2	2	0	0	4
Median	3	3	2	0	4

Table 4
Ns-3.8 Object Oriented Design stats.

	WMC1	WMCv	DIT	NOC	CBO
Mean	10.61	5.43	1.57	0.53	**7.39**
Deviation	19.82	7.91	1.72	3.52	**18.38**
Min	0	0	0	0	0
Max	467	82	5	77	276
Mode	0	0	0	0	1
Median	7	3	1	0	4

- WMC: Weighted methods per class. The sum of a weighting function over the functions of the module. Two different weighting functions are applied: WMC1 uses the nominal weight of 1 for each function, and hence measures the number of functions; WMCv uses a weighting function which is 1 for functions accessible to other modules, 0 for private functions.
- DIT: Depth of inheritance tree. The length of the longest path of inheritance ending at the current module. The deeper the inheritance tree for a module, the harder it may be to predict its behavior. On the other hand, increasing depth gives the potential of greater reuse by the current module of behavior defined for ancestor classes.
- NOC: Number of children. The number of modules which inherit directly from the current module. Moderate values of this measure indicate scope for reuse, however high values may indicate an inappropriate abstraction in the design.
- CBO: Coupling between objects. The number of other modules which are coupled to the current module either as a client or a supplier. Excessive coupling indicates weakness of module encapsulation and may inhibit reuse.

4.5.1. Weighted methods per class: WMC

Among the different weighting functions available to calculate the Weighted Methods per Class metric (WMC) [44], the cccc tool uses WMC1, which assigns weight 1 for each function, measuring the number of functions. WMCv only counts functions which are accessible to other modules, assigning value 0 to private functions.

High values for the WMC metrics mean a high number of methods per class, which can be translated into excessive functionality and complexity per module. A module with a high value for WMC may be susceptible of being split into two or more new modules in order to increase the cohesion of their features.

Ns-3 has a higher value for WMC1, which counts both private and public methods of each class. Regarding WMCv, which only takes into account public methods, both ns-2 and ns-3 have similar values. Thus, the average complexity of the public interfaces of the classes of both simulators are similar. The main difference arises in the field of the private methods, where ns-3 shows a higher degree of complexity.

4.5.2. Depth of inheritance tree: DIT

The Depth of Inheritance Tree (DIT) [44] shows similar figures for both latest releases, with ns-2's classes having a slightly deeper inheritance tree and minor deviation. The graphs below, Figs. 8 and 9, show the general class hierarchy and architecture of both network simulators, you can see that the ns-3 tree is not as deep by design as ns-2's. Thus, the results for DIT metric can be attributed to architectural and design decisions rather than reuse issues. In both cases, the depth of their respective Inheritance Trees is not enough to seriously affect the understanding of the general architecture.

4.5.3. Number of children: NOC

As for the Number Of Children of each module (NOC), both simulators have many classes with few or no child classes, although several classes do have a high number of children revealed by the values of the deviation and maximum of both simulators [44,56]. Again ns-3 shows more content values in terms of number of children.

Observing the Graphical Class Hierarchy [57] included in the official ns-3 documentation and generated for each stable release, the modules with numerous children can be detected easily.

In ns-3 there are two different kinds of modules with high ratios of children. On the one hand, there are entities defined to articulate and create a solid architecture, thereby facilitating the code's reuse, integration and modularity. Classes such as `ns3::empty`, `ns3::AttributeValue`, `ns3::CallbackImplBase` or `ns3::RamdomVariable` are examples of this first type. On the other hand, some entities are closely related to network elements, these can be either real entities or abstractions that may include several network entities. `ns3::Chunk`, `ns3::Header`, `ns3::Channel`, `ns3::Ipv4RoutingProtocol` or `ns3::NetDevice` are representatives of this second group.

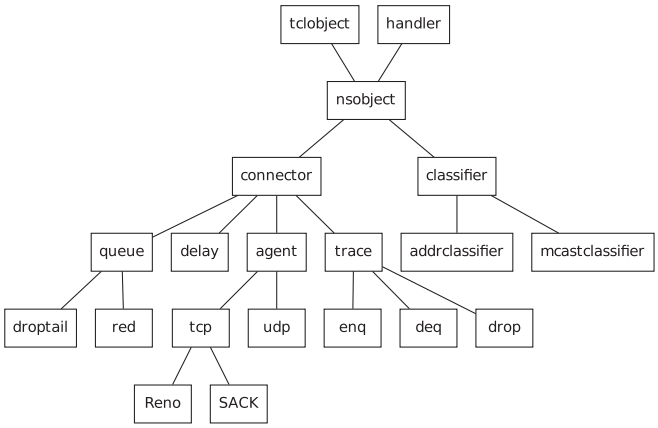

Fig. 8. Simplified class hierarchy tree for ns-2.

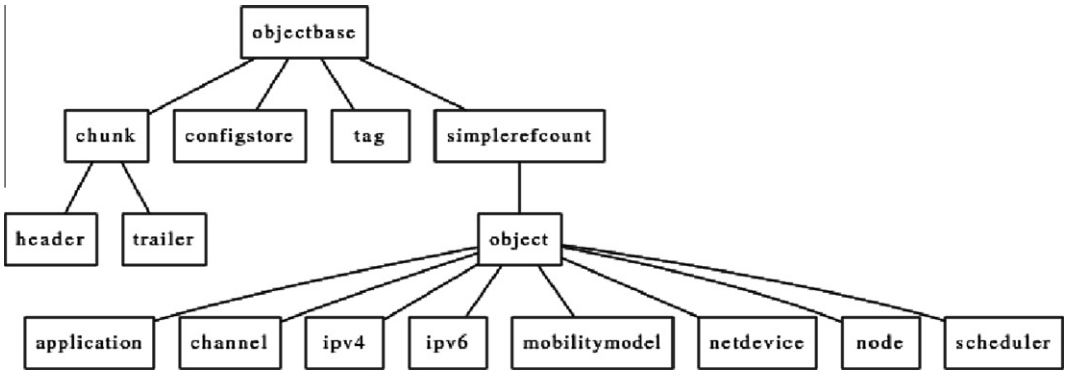

Fig. 9. Simplified class hierarchy tree for ns-3.

On the other hand, a similar Class Hierarchy graphic for ns-2 can be observed in [58]. As stated above, ns-2 presents a deeper inheritance tree, which can be clearly observed in the graphical representation. The strong influence of the dual-language architecture OTcl/C++ is also visible in the class hierarchy graph, with `TclObject` and `TclClass` classes being the parent class of a huge number of modules. Thanks to the class hierarchy graphical representation, you can see that multiple inheritance is used in its design, with all the complexity issues derived from this fact. An example that illustrates the above statement is the `Node` class, which inherits from `RNode`, `nodehead_and ParentNode`. In contrast, the ns-3 graph shows that this technique is not used.

4.5.4. Coupling Between Objects: CBO

Concentrating solely on bare results, ns-3 shows a higher Coupling Between Objects (CBO) value [44,59] that could be interpreted as a sign of excessive coupling between its modules, which would be against ns-3 emphasis on code correction. A more detailed analysis of results shows that the standard deviation for CBO is significantly higher for ns-2 (see bold values in Tables 3 and 4). Ns-3 has certain C++ modules that are the most referenced by others. These highly referenced modules correspond to the main ns-3 abstract entities that articulate its architecture. Most elements and models inherit from them, which increases the overall CBO value, so ns-3's own design is responsible for increasing COB value but this does not mean an implicit weakness in module encapsulation nor does it affect code reuse at all; on the contrary, its design encourages coding generic methods and algorithms as much as possible, making widespread use of templates, software patterns and C++ idioms.

A second reading of the CBO results for ns-3 is possible. After sorting all their modules on the basis of the value of the CBO metric, the modules at the top of the list are shown in Table 5

Many of the modules shown in the top 15 ns-3 elements sorted by CBO values are from classes related to data types such as `Ptr` (memory pointer), `uint32` (32-bit unsigned integer), `bool`, string or vector, as well as auxiliary objects used commonly in Object Oriented Design such as Iterator or Object. However, these types of common modules, with the exception of `bool`, are not present among the ns-2 modules with higher CBO values as shown in Table 6.

The higher values of these modules for the CBO metric can be explained by the different techniques used to ensure code portability (`uint32`, `uintl6`) and make memory managing easier (Ptr and smart pointers). If these generic modules are ignored and a second analysis is performed, the results are in Table 7.

Table 5
Ns-3.8 modules sorted by CBO value, top 15 elements.

Module	WMC1	WMCv	DIT	NOC	CBO
Ptr	11	0	0	0	276
uint32_t	0	0	0	0	270
uint8_t	0	0	0	0	169
ostream	0	0	0	1	148
uint16_t	0	0	0	0	136
Iterator	46	36	0	0	133
bool	0	0	0	0	132
string	0	0	0	0	112
Time	0	0	0	0	108
Object	4	1	2	77	78
Ipv4Address	28	9	0	0	76
Mac48Address	20	15	0	0	76
Header	6	3	2	54	60
vector	0	0	0	1	58

Table 6
Ns-2.34 modules sorted by CBO value, top 15 elements.

Module	WMC1	WMCv	DIT	NOC	CBO
Packet	2	1	1	0	255
Event	1	1	0	6	211
Handler	2	2	0	47	178
TimerHandler	10	8	1	97	99
Agent	42	41	3	49	69
TclObject	0	0	0	40	63
NsObject	12	12	1	9	55
nsaddr_t	0	0	0	0	53
Mac802_15_4	88	51	4	0	49
Node	32	28	2	3	40
RandomVariable	5	5	1	25	39
bool	0	0	0	0	36
SctpAgent	80	80	4	5	32
TcpAgent	64	64	4	12	31
Mac802_11	89	15	4	0	30

Table 7
Ns-3.8 Object Oriented Design stats, ignoring common modules with high values for CBO metric.

	WMC1	WMCv	DIT	NOC	CBO
Mean	10,68	5,46	1,59	0,43	**5,70**
Deviation	19,96	7,88	1,72	2,46	**6,97**
Min	0	0	0	0	0
Max	467	82	5	54	76
Mode	0	0	0	0	1
Median	7	3	1	0	4

Thus, after ignoring some generic modules not related to networking entities or architectural elements, the results obtained for CBO metric for both simulators are similar, with CBO value of 5,70 for ns-3.8 (unlike the original value), while ns-2.34 has a CBO value of 5,56. The profuse use of such generic modules that implement different generic data types such as string or uint32 partially explains the initial results for the CBO metric returned by the analysis tool. This fact plus an architecture where several key entities related to highly referenced abstract networking elements justify the initial CBO values that might be misunderstood as a signal of high levels of coupling between ns-3 modules.

5. Conclusion

A qualitative comparison between ns-2 and ns-3 organization, design and documentation highlights some ns-3 features that overcome certain drawbacks and weak points of ns-2. This fact plus the growing set of networking features and technologies already implemented for ns-3 make it a networking simulation tool suitable for use in many scenarios and contexts hitherto dominated by ns-2.

The quantitative comparison between ns-2 and ns-3 network simulators by statistical analysis of their derivated software metrics reveals several points related with their size, complexity and design.

Ns-2 is portrayed as the mature project that has already fulfilled its aims and has entered into a stagnation phase defined by minor maintenance and bug-fixing releases. In contrast, and judging by its features and metrics concerning size and number of modules, ns-3 is reaching a degree of maturity that makes it suitable for replacing ns-2 in many scenarios and applications.

Due to its design, ns-2 provides a high degree of freedom for coding new models, so the quality of new third-party models depends on their own developers, who are responsible for most of the quality aspects of their code and corresponding documentation. Ns-3 tries to avoid some of ns-2's limitations: for example, it has dropped ns-2's dual-language design, and it emphasizes both code and model structuring as well as encouraging software engineering practices to improve code and documentation maintenance.

The graphs show the different evolution of both developments, with the rapid progress of ns-3 evident and partly explained by its own nature as ns-2's substitute, recycling many of its concepts and ideas, as well as improving several of ns-2's weak points, using a new architectural approach characterized by modularity, source code reuse and software pattern application.

From the point of view of Object Oriented Design, the applied metrics show numerically comparable results, which may lead to a first impression suggesting that there are no important improvements in ns-3 with regard to ns-2. The raw results

are explained taking into account the design characteristics that are responsible for such results, so the numerical results must be accompanied by some extra knowledge about each simulator in order to interpret the results correctly.

Acknowledgment

This work was partially supported by contract Vulcano: TEC2009-10639-C04-02.

References

[1] M.A. Rahman, A. Pakštas, F.Z. Wang, Network modelling and simulation tools, Simulation Modelling Practice and Theory 17 (6) (2009) 1011–1031.
[2] M.S. Obaidat, N.A. Boudriga, Fundamentals of Performance Evaluation of Computer and Telecommunication Systems, John Wiley & Sons, Inc., 2010.
[3] Educational Use of Ns-2. <http://www.isi.edu/nsnam/ns/edu/index.html>.
[4] Ns-3 Overview. <http://www.nsnam.org/docs/ns-3-overview.pdf> (October 2009).
[5] E. Weingärtner, H. Lehn, K. Wehrle, A performance comparison of recent network simulators, IEEE International Conference on Communications, 2009. ICC '09, pp. 1–5, 2009.
[6] M.S. Obaidat, G. Papadimitriou, Applied System Simulation: Methodologies and Applications, Springer, 2003.
[7] J.L. Font, P. Iñigo, M. Domínguez, J.L. Sevillano, C. Amaya, Architecture, design and source code comparison of ns-2 and ns-3 network simulators, in: 13th Communications & Networking Simulation Symposium, CNS-10, Orlando, FL, pp. 29–36, 2010 (Best Paper Award).
[8] Ns-3 Supported OS. <http://www.nsnam.org/getting_started.html>.
[9] Ns-3 Tutorials: Development Environment. <http://www.nsnam.org/docs/tutorial/tutorial_9.html>.
[10] Ns-3 Wiki: Installing ns-3 on Mac OSX. <http://www.nsnam.org/wiki/index.php/HOWTO_get_ns-3_running_on_Mac_OS_X_(10.5.2_Intel)>.
[11] CVS Project. <http://www.cvshome.org/>.
[12] Mercurial Project. <http://mercurial.selenic.com/>.
[13] The Network Simulator: Building Ns. <http://www.isi.edu/nsnam/ns/ns-build.html>.
[14] Ns-3 Tutorials: Getting Started. <http://www.nsnam.org/docs/release/tutorial.html#Getting-Started>.
[15] Tcl Developer Site. <http://www.tcl.tk>.
[16] OTcl Site. <http://otcl-tclcl.sourceforge.net/otcl/>.
[17] Ns-2 Debian Package. <http://packages.debian.org/sid/ns2>.
[18] Ns-3 Debian Package. <http://packages.debian.org/sid/ns3>.
[19] J. Chung, M. Claypool. NS by Example: Extending NS. <http://nile.wpi.edu/NS/>.
[20] Ns-3 User FAQ: WAF (build process). <http://www.nsnam.org/wiki/index.php/User_FAQ#WAF_.28build_process.29>.
[21] Waf Project. <http://code.google.com/p/waf/>.
[22] M. Lacage, Experimentation with ns-3, Trilogy Summer School. <http://www.nsnam.org/tutorials/trilogy-summer-school.pdf> (27.08.09).
[23] Ns-3 User FAQ: Python Bindings. <http://www.nsnam.org/wiki/index.php/User_FAQ#Python_bindings>.
[24] M. Lacage, T. Henderson. Yet another network simulator, in: WNS2 '06: Proceeding from the 2006 Workshop on ns-2: the IP Network Simulator, 2006.
[25] The ns Manual: 5.1 Node Basics. <http://www.isi.edu/nsnam/ns/doc/node40.html>.
[26] The ns Manual: 6.1 Simple Links. <http://www.isi.edu/nsnam/ns/doc/node57.html>.
[27] The ns Manual: 12.1 A Protocol-Specific Packet Header. <http://www.isi.edu/nsnam/ns/doc/node128.html>.
[28] Ns-2 Release Archive. <http://www.isi.edu/nsnam/dist/>.
[29] Ns-3 Manual: 4.1 Object Model. <http://www.nsnam.org/docs/release/manual.html#Object-model>.
[30] Ns-3 Tutorial: 4.1 Key Abstractions. <http://www.nsnam.org/docs/release/tutorial/tutorial_17.html#Key-Abstractions>.
[31] Marc Greis's Tutorial, VINT Group, VII.3. Necessary Changes. <http://www.isi.edu/nsnam/ns/tutorial/nsnew.html#third>.
[32] Doxygen Source Code Documentation Generator Tool. <http://www.stack.nl/dimitri/doxygen/>.
[33] Ns-3 Doxygen Documentation. <http://www.nsnam.org/doxygen-release/index.html> (October 2009).
[34] Ns-3 Release Archive. <http://www.nsnam.org/releases/>.
[35] CCCC: C and C++ Code Counter. <http://cccc.sourceforge.net/>.
[36] Network Simulator Cradle. <http://www.wand.net.nz/stj2/nsc/>.
[37] Ns-2.34 Changelog. <http://www.isi.edu/nsnam/ns/CHANGES.html>.
[38] Ns-2 Older Releases. <http://www.isi.edu/nsnam/dist/>.
[39] Ns-2 Download Site. <http://www.isi.edu/nsnam/ns/ns-build.html>.
[40] Ns-3 Download Site. <http://www.nsnam.org/download.html>.
[41] Ns-3 Project Maintainers. <http://www.nsnam.org/maintainers.html>.
[42] Ns-3 Current Development. <http://www.nsnam.org/wiki/index.php/Current_Development>.
[43] Ns-3 Official Blog. <http://nsnam.blogspot.com>.
[44] T. Littlefair, An Investigation into the role of Software Metrics in Software Quality Improvement. PhD research project, Edith Cowan University, Australia, 1999.
[45] Ns-3 Software Architecture, Ns-3 Project. <www.nsnam.org/docs/design.pdf>.
[46] Ns-3 Wiki: Ns-3 Current Development. <http://www.nsnam.org/wiki/index.php/Current_Development>.
[47] G.A. Wainer, Discrete-Event Modeling and Simulation, Taylor & Francis Group, LLC, 2009.
[48] T.J. McCabe, A complexity measure, IEEE Transactions on Software Engineering SE-2 (4) (1976) 308–320.
[49] S. Counsell, E. Mendes, S. Swift, Comprehension of object-oriented software cohesion: the empirical quagmire, in: 10th International Workshop on Program Comprehension, 2002, pp. 33–42.
[50] T.J. McCabe, A.H. Watson, Software complexity, Crosstalk 7 (12) (1994) 5–9.
[51] C. Stein, G. Cox, L. Etzkorn, Exploring the relationship between cohesion and complexity, Journal of Computer Science 1 (2) (2005) 137–144.
[52] T.M. Khoshgoftaar, J.C. Munson, Predicting software development errors using software complexity metrics, IEEE Journal on Selected Areas in Communications 8 (2) (1990) 253–261.
[53] B. R Basili, B.T. Perricone, Software errors and complexity: an empirical investigation, Communications of the ACM 27 (1) (1984) 42–52.
[54] Ns-3 Coding Style. <http://www.nsnam.org/codingstyle.html>.
[55] R.S. Pressman, Software Engineering, a Practitioner's Approach, 5th ed., McGraw-Hill, 2000.
[56] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.
[57] Ns-3 Graphical Class Hierarchy. <http://www.nsnam.org/doxygen-release/inherits.html>.
[58] Ns-2 Graphical Class Hierarchy. <http://www.auto-nomos.de/ns2doku/inherits.html>.
[59] R. Harrison, S. Counsell, R. Nithi, Coupling metrics for object-oriented design, in: Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International, 1998, pp. 150–157.