Architecture, design and source code comparison of 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
Email: {juanlu,pabloinigo,mdominguez,sevi,camaya}@atc.us.es

Keywords: Discrete-event simulation, design, ns-2, ns-3, software metrics

Abstract

Ns-2 and its successor ns-3 are discrete-event simulators. Ns-3 is still under development, but offers some interesting characteristics for developers while ns-2 still has a big user base. This paper remarks current differences between both tools from developers point of view. Leaving performance and resources consumption aside, technical issues described in the present paper might help to choose one or another alternative depending of simulation and project management requirements.

1. INTRODUCTION

Simulation is a key tool in networking research due to its inherent advantages over testing with real hardware. Matters as budget and cost of time make simulation attractive for research, although flexibility, scalability and almost non-cost expansion are also valuable advantages over other testing methods. Network simulators allow to implement and study different network entities in a simulated environment, giving a high degree of flexibility to test new protocols, technologies, physic models and topologies.

Ns-2, a discrete-event network simulator, has been the de facto standard for the last decade, being widely used in academic research. Ns-3 [3] [**?**] has been conceived as ns-2 successor, its developers have tried to solve or mitigate many of the ns-2 well-known drawbacks as well as to apply new concepts, such as validation and software engineering techniques, in order to produce a more reliable simulation tool to support academic and industrial research [5].

Other technical papers have already addressed ns-2 and ns-3 comparison [6], among other network simulators, evaluating their core performance and scheduling capabilities and avoiding to focus on any specific simulator feature in order to make a comparison of very different pieces of software as fair as possible. This paper aims to make a technical comparison between ns-2 and ns-3, emphasizing on their architecture, design, advantages and drawbacks from the point of view of developers interested in network simulation. Due the influence of ns-2 over ns-3 design and development, it is possible to elaborate a thorough analysis focused on technical and design characteristics that would not be possible to carry on with other simulators quite different in nature and aims.

Subject such as host operating system integration, source code hierarchy, generation of binaries, portability or structuring and documenting policies are discussed on the present paper, remarking pros and cons of both simulators in development, code maintenance and quality terms. Some software metrics has been included in order to show both projects evolutions in terms of size, documentation and source code coupling.

2. SYSTEM INTEGRATION

Ns simulators family has its roots in Unix-like environments, being developed using typical Unix languages and tools. Nowadays, GNU/Linux is one of the main development platforms for both ns-2 and ns-3 [7]. Although they have been ported to other operating systems such as Windows/Cygwin [8], *BSD or Mac OSX [9], working with GNU/Linux makes easier to install, update and maintain instances of both simulators.

2.1. Distribution

Both ns-2 and ns-3 are mainly conceived to be distributed in source code form. Since users have access to source code, they can modify and extend their features and optimize their binaries as needed. 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[10], an emerging distributed revision control tool, to manage its.

Ns-2 has several software dependences, 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 [12]. Depending on very specific, and sometimes, obsolete library versions becomes an important drawback when installing ns-2. This problem is mitigate by the project maintainers through packaging the whole dependencies in a single installation package, aka *all-in-one*. This

package contains both ns-2 and libraries source code, configuration files and tools needed to compile the whole project. This installation alternative is better than manually fetching source code from ns-2 CVS repository and trying to solve problems with library paths and compilation errors by hand.

As well as ns-2, ns-3 offers an *all-in-one* installation package [13] which only contains ns-3 source code. Thanks to the easy to install its software dependencies, fetching and a ns-3 from its Mercurial repository is a good and easy way for developers to stay synchronized with the main ns-3 development branch.

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 with recent gcc versions. Apart from the toolchain, ns-2 and ns-3 have several library dependences. Main ns-2 required dependencies include:

- 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) [14], although ns-2 developers state that the simulator has only been tested against 8.4.14.
- OTcl: a Tcl object-oriented extension [15], it is used as ns-2 scripting language for writing simulations. The last release dates from 2007-03-10 and for the last five 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 little diffusion and use.
- TclCL: is Tcl/C++ interface that allows the creation of OTcl wrappers for C++ code. The development of TclCL is tightly related with OTcl. As well as OTcl, TclCL is not a widespread tool and it cannot be found in most software repositories of the main GNU/Linux distributions.

Ns-3 requires external libraries too, 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 rest of the dependencies will be installed optionally in order to enable some specific features. Software such as valgrind, sqlite or GNU Scientific Library are examples of these optional dependencies. All of them are easy to find in the repositories of mainstream distributions like Debian, Ubuntu or Fedora [7]. Packaging tools make software installation much easy and neat by automatically resolving dependencies and configuring library paths. In addition, there are no major issues related with using too new library versions, which are soon tested against current ns-3 code.

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 needed libraries and tools, so users have a higher grade of flexibility to solve software dependencies independently of their specific host operating system. This advantage is almost mandatory for a software that is still under active development that is distributed all over the world. By contrast, Ns-2 shows signs of aging and its development seems to be stuck, the project itself cannot spend too much resources on testing and updating external software dependencies so ns-2 finally relies on mostly obsolete versions which means a major drawback for installation and maintenance process.

2.3. Building the source code

Ns-2 uses a classic Unix configuration and building process based on the `make` building tool. In order to add new modules to ns-2 or to change current configuration, the user can edit himself the main Makefile of the project [16] or define his own makefile in order to compile and generate the associated libraries. The core and modules building process can generate a monolithic executable called `ns` which contains the simulator functionalities as well as all the current models distributed with ns-2. In order to avoid this situation, the developer can define his own building rules and generate a ns-2 standard core and independent libraries with the new models. Simulations are just OTcl scripts, but depends on `ns` binary to run. Scripts are passed as parameters to `ns`, which act as as an script interpreter. On the other hand, if a separated makefile is written, independent binary code from ns-2 core will be obtained.

Ns-3 supports Python scripting, although writing simulations in this language is not mandatory and Python support and its wrappers can be optionally disabled. Ns-3 also relies on Python for configuring and compiling its own source code [17]. Instead of using `make` tool as ns-2 does, ns-3 has adopted `waf` as a build automation tool [18], which has Python as single external dependence. As well as Python language itself, `waf` is portable. It allows to compile only modified source code files, ignoring the untouched ones. Rebuilding a relatively simple class takes only between 1 and 2 seconds when using the same AMD Quad Core, 4GB 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 rather a shared library than a traditional executable as ns-2 one is. `Libns3.so` contains the simulation routines as well as all the ns-3 standard modules.

3. PROJECT DESIGN AND SOURCE CODE PROPERTIES

In this section it is discussed both project general architecture, source code structuring as well as other auxiliary aspects for developers such as documentation and portability. These issues are approached qualitatively, being quantitative metrics left for the next section.

3.1. Dual-language architecture

The wide use of Tcl and its language extensions in ns-2 responds to a past context in which compiled languages such as C++ where 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 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 invokes the functionalities coded in C++ thanks to software wrappers [15]. Despite of reducing compilation times, the dual-language nature adds extra complexity to ns-2 use and development. It can be difficult to new developers to identify which parts have to be coded in C++ and which ones in OTcl language when a coding a new model.

Nowadays, current hardware is powerful enough to handle with compiled languages like C++, consuming negligible compilation times in most cases, so the advantages derived from using a scripting languages like OTcl have partially vanished.

Ns-3 has simplified its design choosing C++ as single development language for its models, simulations can be coded in C++ too. Ns-3 has not totally dropped the use of scripting languages for coding simulations, but has changed the way they are used into the project [19]. Compilation times are no more an issue due to the power of current hardware, so advantages from using scripting languages derive from other technical reasons such as portability, productivity or an easier syntax. Ns-3 users can optionally choose 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 more friendly syntax.

3.2. Architecture and code organization

Ns-2 defines a very basic architecture, making only distinction between 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 dependences are difficult to follow easily [20].

Ns-2 differentiates between the network topology and the agents that run over it and generate or consume data traffic[21]. The topology is formed by generic nodes and the links between them. Also, these links can have special objects associated that define the link behavior [22], 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. The model developer is responsible for assuring the agent layer will be able to exchange data packets with the network topology layer, so he has to design his own data packets if necessary and register them as new types so that ns-2 would be able to handle them. Packets registration requires the user to change some source code files of ns-2 core and to rebuild the whole simulator [23], which is a quite 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 accuracy ones. Thus, simulation results must be analyzed carefully, depending on the level of abstraction applied, some simulation results can considerably diverge from real-world ones [5].The ns-2 relaxed hierarchical organization is reflected on 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 2.26 releases, maintainers have made efforts in order to maintain a proper source code structure [24].

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 [31].

- Node: represents the basic computing device. It gathers the network functionality and interacts with other nodes by the communication channel. Nodes require net devices in order 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 media to interconnect nodes and to allow 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 rest of auxiliary tools, building scripts, documentation and examples. The `src` folder is also subdivided, attending to the above classification of abstract entities. Thus, all the models related with network devices can be found into `src/devices`. This hierarchical organizations allows developers to easily navigate through the simulator code. Following the above scheme, a ns-3 programmer is encouraged to define several entities in his abstract model in order to fit his code in the global simulator structure, assuring a common model skeleton that makes the overall code more flexible and adaptable, being easy to refine and reuse it in the future. Due to the `src` folder has been conceived just for containing the standard ns-3 modules, which are maintained, supported and documented by 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.

Comparing both ns-2 and ns-3 partial class hierarchy, it can be noted the more intuitive structure shown by ns-3 source code.

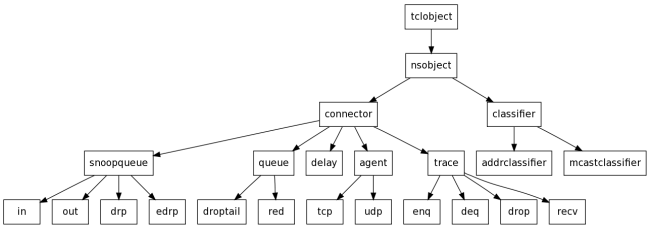

Figure 1. Partial ns-2 Class Hierarchy

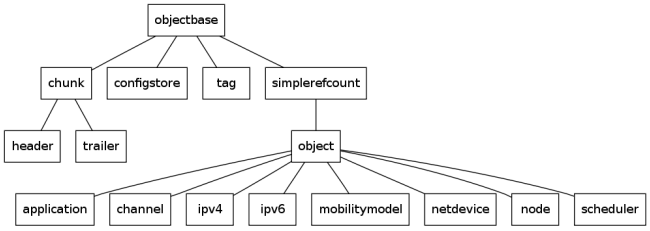

Figure 2. Partial ns-3 Class Hierarchy

Due to space restrictions, partial graphs have been attached. For a more complete ns-2 and ns-3 class hierarchy models, visit [30] and [29].

3.3. Simulation portability

Ns-2 simulations are initially portable due the fact they are simple OTcl scripts. There is no problem to launch a ns-2 simulation in a different ns-2 instance, independently of the target architecture of the `ns` binary that will run the script. This is always true whenever the ns-2 script only uses 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 part of ns-2 project and the building process is modified in order to include these new models into the core itself, as an unique monolithic entity; it will need a custom `ns` executable, generated from the standard ns-2 source code plus the new model code. Thus, the ns-2 design may attach a simulation that uses custom models to the specific ns-2 instance that contains them, losing any kind of advantage related with portability due to the need of a custom `ns` interpreter. This situation can be avoided if the developer defines and generates his independent external libraries, keeping a standard ns-2 core. In this case, his simulations must be accompanied by their complementary external libraries.

Ns-3 design overcomes the above possible 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 as 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 with 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. Due to documenting ns-2 can be a double and non-automated task, recording information on source code files and on the web, some source files may lack proper documentation that can only be found on the project web, and vice versa[1]. Synchronization between code under development and its associated code is in most cases a manual process. A first look at the ns-2 official web site shows its documentation structure, divided into: core documentation, it includes several FAQs[25], third-party tutorials and an extensive reference[26]; development help, which gathers messages from several mailing lists[27] as well as the change history from the CVS repository[28]. A C++ class hierarchy is present too[29].

Ns-3 integrates documentation in source code files by using Doxygen [32], an open-source multi-language documentation generator, which parses and extract ns-3 documentation directly from source code. It allows to define some format aspects of the documentation such as defining parameters, return values, links to others pieces of documentation, etc.

Besides, 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. Starting from the bare source code and the comments formatted in the Doxygen way, all the documentation related content can be extracted in order to generate a complete user manual. This is the way the original handbook for developers from the ns-3 project is generated and kept to date [2]. This manual is an extensive and exhaustive enumeration of code elements that includes detailed description of C++ class hierarchy, programming interfaces and module contents.

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

4. QUANTITATIVE ANALYSIS OF SOURCE CODE

C++ source code files from several ns-2 and ns-3 previous and current releases have been analyzed in order to extract basic software metrics (ns-2: from 2.0 to 2.34, 12 years, ns-3: from 3.0 to 3.6, 3 years) [24] [33]. In one hand, ns-3 is shorter lived than ns-2, but it has benefit from ns-2 ideas and experience, so it cannot be considered a totally from scratch development. In the other and, ns-2 has experimented different stages along its long existence and nowadays it can be considered at the end of its life cycle.

4.1. Roadmap and releases

Several ns-3 and ns-2 releases have been analyzed in order to evaluate their respective evolution by extracting some basic software metrics. Both current and previous releases has been fetched from their official source code repositories and archives respectively.

Ns-2 started as variant of the REAL network simulator in 1989. Today, its changelogs reflects 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.

Ns-3 project started in 2006 and was projected to take four years. Nowadays is still under active development, having several full-time hired maintainers [35]. Due it depends on contributed modules, it is difficult to define an exact project roadmap [36].

4.2. Software metrics

The *cccc* tool, C and C++ Code Counter [38], is an open-source software metrics tool that has been chosen to perform the code analysis. It can only parses C++ files (`*.cc` and `*.h`), so code related with Tcl/OTcl, Python and other script languages has been ignored. The next stats belong to the simulator core and default models of both ns-2 and ns-3.

Table 1. Ns-2 versions by release date [34]

	Version	Release date
1	ns-2.0	July 25, 1997
2	ns-2.1b	July 3, 2002
3	ns-2.26	February 26, 2003
4	ns-2.27	January 18, 2004
5	ns-2.28	February 3, 2005
6	ns-2.29	October 19, 2005
7	ns-2.30	September 26, 2006
8	ns-2.34	June 17, 2008

Table 2. Ns-3 versions by release date [37]

	Version	Release date
1	ns-3.0.1	March 31, 2007
2	ns-3.0.13	June 6, 2008
3	ns-3.1	Jul 1, 2008
4	ns-3.2	September 25, 2008
5	ns-3.3	December 28, 2008
6	ns-3.4	April 6, 2009
7	ns-3.5.1	September 23, 2009
8	ns-3.6	October 21, 2009

Although it is not possible to directly compare the overall results due the different amount of features that ns-2 and ns-3 implement, it is useful to analyze and compare averages in order to make a more realistic and fair code comparison. The following procedural software metrics have been extracted from their respective releases:

- 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

- IF4v: Information Flow measure. Measure of information flow between modules suggested by Henry and Kafura. The analyzer makes an approximate count of this by counting inter-module couplings identified in the module interfaces. IF4v is calculated using only relationships in the visible part of the module interface.

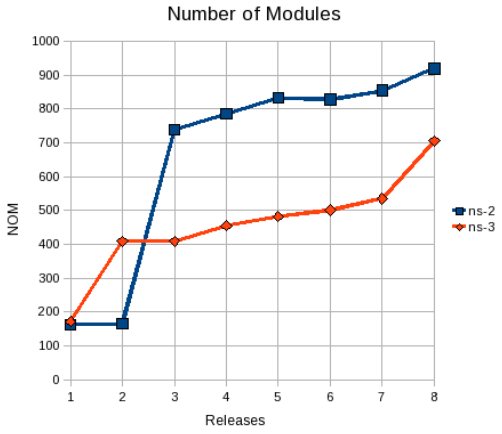

Figure 3. Total number of modules

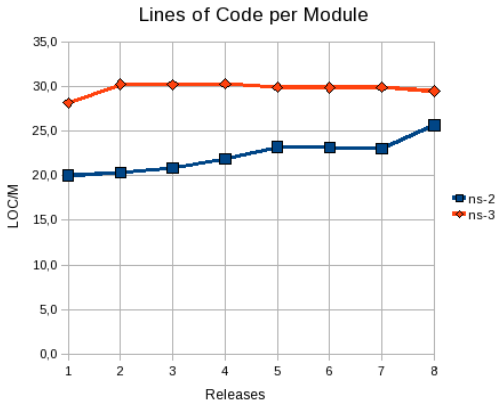

Figure 4. Lines of code per module

Apart from the above metrics, a more detailed analysis of the last ns-2 and ns-3 releases has been made (ns-2.34 and ns-3.6 respectively). The following Object Oriented Design metrics have been used:

- 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.

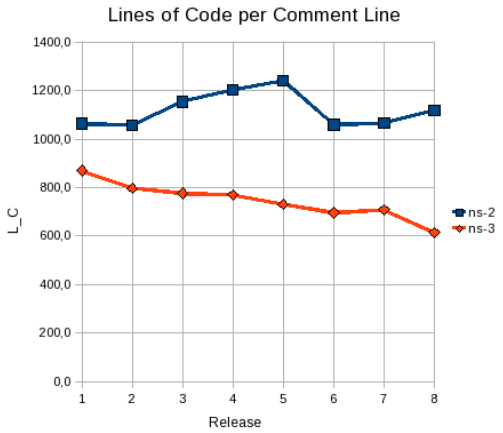

Figure 5. Number of lines of code per comment line

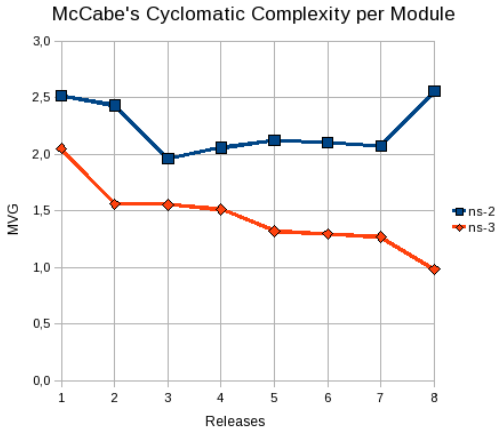

Figure 6. McCabe's Cyclomatic Complexity per Module

- 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.

Paying attention only to the raw results, ns-3 shows a

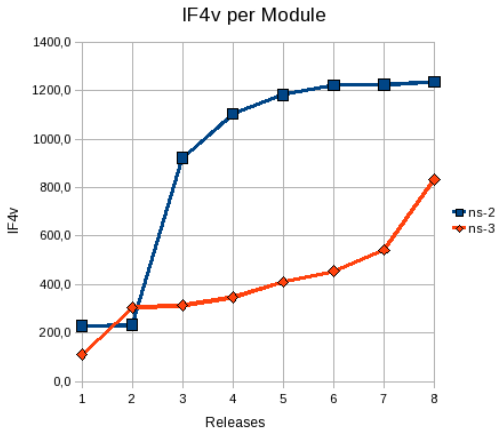

Figure 7. IF4v per Module

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	**13,9**
Min	0	0	0	0	0
Max	465	300	7	97	255
Mode	2	2	0	0	4
Median	3	3	2	0	4

higher CBO value that could be interpreted as a sign of excessive coupling between its modules, which would be against ns-3 emphasis on code correction.

This specific fact does not really mean a true high degree of coupling. A closer study of the statics shows that the standard deviation for CBO is higher than ns-2 as well as ns-3 mode is significantly lower. Ns-3 has a few C++ modules that are the most referenced by others. These highly referenced ones correspond with the main ns-3 abstract entities that articulate its architecture, such as device, node or channel class . Most elements and models inherit from them, which increases the overall CBO value. If these highly referenced classes would be ignored during the analysis, the rest of the ns-3 classes showed a very low CBO value, which it is consistent with the wide use of templates, software patterns and C++ idioms that aim to improve the overall code quality. This second CBO value, ignoring singularities and extreme values, should be considered in order to obtain a more realistic metric about coupling in the context of ns-3 project.

5. CONCLUSION

Ns-2 its a quite mature project that has already matched its aims. Nowadays its development is stuck and only minor maintenance versions are released, as well as it has as software dependencies some obsolete or not too widespread libraries and tools. Due to its design, ns-2 gives a high degree of freedom to code new models, so quality of new third-party ones depends on their own developers, who are responsible of most of the quality aspects of their code and their corresponding documentation. Ns-3 tries to save some of ns-2 limitations, for example, dropping ns-2 dual language design, and it emphasizes both code and model structuring as well as encourages software engineering practices in order to improve code and documentation maintenance. Installation and dependencies are well integrated into most commons host operating systems as well as source code building process is neat and well automated. Ns-3 design tries to keep its core and users simulations and code separated, giving proper building automation tools and trying to guarantee a minimum quality standards for all third-party developers who follow their recommendations. By comparing some software metrics, it can be realized ns-3 has experimented a fast paced development, being close to be a fully viable ns-2 successor in the very near future. Most of it modules show a balanced level of complexity and coupling. In conclusion, ns-3 can provide new and interesting advantages over ns-2 for developers seeking to create new network models, and it is ready to evolve and grow in a sustainable way as a collaborative project.

Table 4. Ns-3.6 Object Oriented Design stats

	WMC1	WMCv	DIT	NOC	CBO
Mean	10,07	5,21	1,13	0,48	**6,83**
Deviation	20,44	7,63	1,34	2,71	**15,2**
Min	0	0	0	0	0
Max	447	73	4	54	222
Mode	0	0	0	0	1
Median	6	3	0	0	4

6. ACKNOWLEDGEMENTS

This work was supported by contracts TIN2006.15617.C03.03, AmbienNet: Ambient Intelligence Supporting Navigation for People with Disabilities; and P06-TIC-2298, SemiWheelNav: External sensing based semiautonomous wheelchair navigation.

REFERENCES

[1] Ns-2 documentation. http://www.isi.edu/nsnam/ns/

[2] Ns-3 Doxygen documentation. http://www.nsnam.org/doxygen-release/index.html

[3] Educational use of ns-2. http://www.isi.edu/nsnam/ns/edu/index.html

[4] Ns-2 module contributions. http://www.isi.edu/nsnam/ns/ns-contributed.html

[5] Ns-3 overview. http://www.nsnam.org/docs/ns-3-overview.pdf

[6] E. Weingärtner, H.Lehn, K.Wehrle. A performance comparison of recent network simulators. IEEE International Conference on Communications, 2009. ICC '09. 14-18 June 2009 Page(s):1 - 5.

[7] Ns-3 supported OS. http://www.nsnam.org/getting_started.html

[8] Ns-3 tutorials: Development environment. http://www.nsnam.org/docs/tutorial/tutorial_9.html

[9] 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)

[10] Mercurial project. http://mercurial.selenic.com/

[11] CVS project.http://www.cvshome.org/

[12] The Network Simulator: building Ns. http://www.isi.edu/nsnam/ns/ns-build.html

[13] Ns-3 tutorials: Getting started. http://www.nsnam.org/docs/release/tutorial.html#Getting-Started

[14] Tcl Developer Site. http://www.tcl.tk

[15] OTcl site. http://otcl-tclcl.sourceforge.net/otcl/

[16] J. Chung,M. Claypool. NS by example: Extending NS. http://nile.wpi.edu/NS/

[17] Ns-3 User FAQ: WAF (build process) http://www.nsnam.org/wiki/index.php/User_FAQ#WAF

[18] Waf project. http://code.google.com/p/waf/

[19] Ns-3 User FAQ: Python bindings. http://www.nsnam.org/wiki/index.php/User_FAQ#Python

[20] M. Lacage, T.Henderson. Yet another network simulator. WNS2 '06: Proceeding from the 2006 workshop on ns-2: the IP network simulator, 2006.

[21] The ns manual: 5.1 Node basics. http://www.isi.edu/nsnam/ns/doc/node40.html

[22] The ns manual: 6.1 Simple links. http://www.isi.edu/nsnam/ns/doc/node57.html

[23] The ns manual: 12.1 A Protocol-Specific Packet Header. http://www.isi.edu/nsnam/ns/doc/node128.html

[24] Ns-2 release archive. http://www.isi.edu/nsnam/dist/

[25] Ns-2 FAQs. http://www.isi.edu/nsnam/ns/ns-faq.html

[26] Ns Manual, formerly called ns Notes and Documentation. http://www.isi.edu/nsnam/ns/ns-documentation.html

[27] Ns related mailing lists.http://www.isi.edu/nsnam/ns/ns-lists.html

[28] Ns-2 CVS History. http://cvs.sourceforge.net/viewcvs.py/nsnam/

[29] Ns-2 C++ Class Hierarchy. http://nile.wpi.edu/NS/components.html

[30] Ns-3 C++ Class Hierarchy. http://www.nsnam.org/doxygen-release/inherits.html

[31] Ns-3 Manual: 4.1 Object Model. http://www.nsnam.org/docs/release/manual.html#Object-model

[32] Doxygen Source code documentation generator tool. http://www.stack.nl/ dimitri/doxygen/.

[33] Ns3 release archive. http://www.nsnam.org/releases/

[34] Ns-2.34 Changelog. http://www.isi.edu/nsnam/ns/CHANGES.html

[35] Ns-3 project maintainers. http://www.nsnam.org/maintainers.html

[36] Ns-3 Current Development. http://www.nsnam.org/wiki/index.php/Current_Development

[37] Ns-3 official blog. http://nsnam.blogspot.com

[38] CCCC: C and C++ Code Counter. http://cccc.sourceforge.net/

[39] T. Littlefair. An investigation into the role of Software Metrics in software quality improvement. PhD research project, Edith Cowan University, Australia, 1999.