ROMA TRE DIA

Università degli Studi di Roma Tre
Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79 – 00146 Roma, Italy

Test Driven Network Deployment

Maurizio Pizzonia[1] and Stefano Vissicchio[1]

RT-DIA-143-2009 **March 2009**

(1) Dipartimento di Informatica e Automazione,
Università di Roma Tre,
Rome, Italy.
pizzonia@dia.uniroma3.it
ste.vissicchio@gmail.com

This work is partially supported by the Italian Ministry of Research, Grant number RBIP06BZW8, FIRB project "Advanced tracking system in intermodal freight transportation" and under Project "MAINSTREAM: Algoritmi per strutture informative di grandi dimensioni e data streams", MIUR Programmi di ricerca di Rilevante Interesse Nazionale.

ABSTRACT

To meet the changing needs of customers and users, computer networks normally undergo frequent, although usually inexpensive, reconfigurations.

However, it is not guaranteed that a network still meets all the previously implemented requirements after a change, since, in complex systems, unexpected side effects can easily arise. We believe that current practice in networking does not provide enough support for the verification of the adherence of an operational network to its requirements, after a change.

We envision a new approach. Our *Test Driven Network Deployment* methodology relies on a deep automation of the testing activity, which allows to simplify change management and better support the evolution of the network. Our proposal is conceived to have a limited impact on the budget of a project and it is suitable to be incrementally adopted.

The methodology we propose is inspired by similar approaches widely adopted in software engineering, in which common practice enables engineers to successfully deal with the continuous adaptation of software.

In this paper, we define the founding principles of the Test Driven Network Deployment methodology, we discuss its affordability, and the features of its automatic tests illustrating them with examples. We also provide a number of requirements and design principles for tools that should support our methodology, laying the basis for their development and for future experimental work in productive environments.

1 Introduction

We believe that it is interesting to consider engineering fields as roughly divided into two categories, *bulk* and *lite*, according to how costs are distributed over the product lifecycle. In bulk engineering fields, the realization phase is much more expensive than the analysis and design phases, hence, design choices are tightly constraining and waterfall methodologies are the most appropriate [47]. For example, civil engineering, mechanics, and microelectronics, belong to this category. In lite engineering fields, the cost of the realization phase is less than or comparable to the cost of the analysis and design phases and it is generally accepted to partially modify what has been realized. This encourages the adoption of iterative methodologies, where the evolution of the project encompasses several iterations. Iterations are normally short with respect to the duration of the entire project and each of them includes analysis, design, and realization phases [14]. For example, software engineering belongs to this category. The network engineering field lies somewhere in the middle but, with respect to many aspects, belong to the second category.

In practice, network engineers and administrators are used to often perform reconfigurations to modify the behavior of the network and implement new requirements, even during operation. However, changing a part of a network can affect other parts of it. This is because computer networks, especially large-scaled highly engineered ones, are complex systems, made of components that are deeply interconnected, and their continuous evolution can make hard the comprehension of their behavior [16, 53]. As a result, changes in networks [38, 41], as in other complex systems [19, 25, 26, 44], can propagate with unexpected non-trivial side effects.

This may lead to unmet requirements or failures and errors which can also be widespread and latent. The relevance of this problem is largely recognized, especially for configuration errors [4, 15, 33, 3, 41, 40].

In our opinion, current testing practice and methodologies (e.g., [43, 28, 51, 36, 6, 46]) do not conveniently support operators in detecting side effects of changes. As a consequence, operators and administrators cannot easily assess if the production network still conforms to previously implemented requirements after a change.

Current practice grounds on the use of testbeds and monitoring tools. Testbeds are prototype environments which are commonly exploited in pre-deployment phases for validating analysis and design choices. The task of verifying the behavior of a production network, on the contrary, mostly relies on the adoption of a lightweight monitoring approach, which is based on simple checks of availability of hosts and servers. Manual in-depth testing takes too much resources and is prone to the risk of further mistakes that can impact both the correctness of tests and the behavior of the whole network. For this reason, in-depth testing is rarely performed on operational networks.

In this paper, we propose a novel methodology, called *TDND – Test Driven Network Deployment*, which aims at facilitating timely detection of errors and harmful side effects of changes. For this purpose, TDND prescribes to execute in-depth tests of requirements during and right after the deployment of each change. The Test Driven Network Deployment methodology is based on a deep automation of the testing activity that enables to frequently execute large sets of tests in an inexpensive way. We believe that TDND can be incrementally introduced in a project since even a small effort, spread over time, can lead to large effectiveness. TDND is mainly complementary to current practice and testing methodologies, and it is related to what is usually called *regression testing*, which, however, is not extensively adopted in practice due to its cost.

Our methodology is inspired by current successful practice in software engineering, a discipline which is similar to networking with respect to continuous evolution of products. Namely, in software engineering, testing activity is usually performed often, in-depth, and in an automated way, within a methodological approach called Test Driven Development [8] which also suggest the usage of convenient standard tools, like JUnit [12]. This practice is a fundamental part of a wider methodology called Extreme Programming (XP) [11, 10] which aims at "embracing change" and has been demonstrated effective in many real life experiences [27, 24, 48, 31].

The main contribution of this paper is the description of the principles of TDND and the discussion of its relationships with both well known network testing approaches and the software testing practice it is inspired by. We also describe the network tests used in the methodology along with their features, which we consider necessary to make TDND affordable, and illustrate them with examples. Further, we provide a number of requirements and design principles for tools that should support our methodology. As a follow-up of this paper, we plan to develop effective testing tools and experiment with them in real projects. We expect that these experiences will suggest improvements and refinements of the methodology,

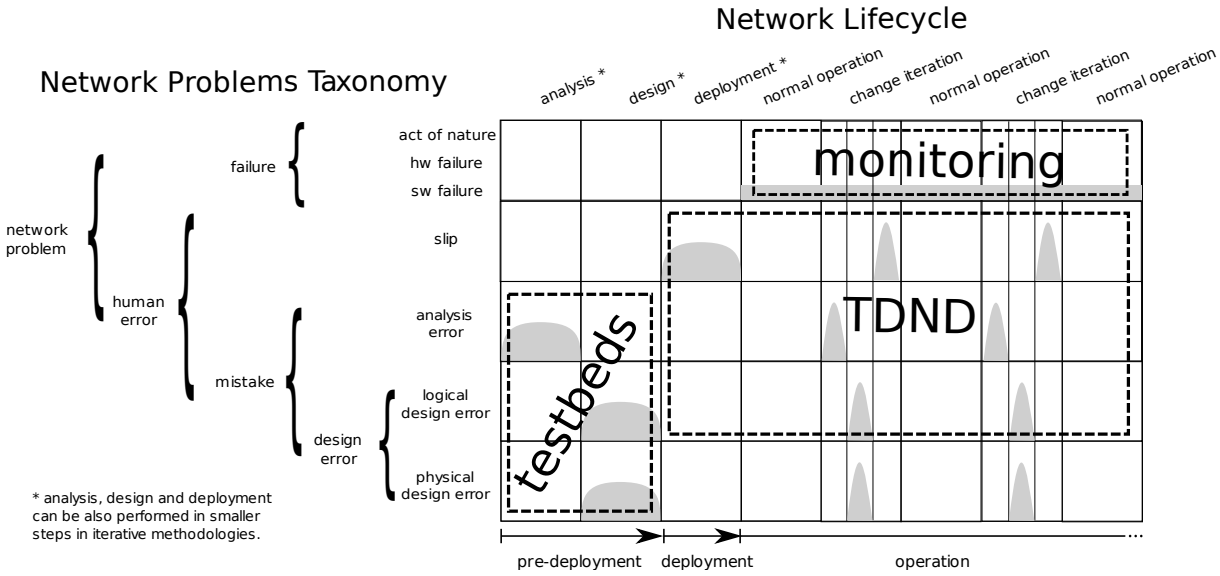

Figure 1: On the left, a taxonomy of network problems. On the right, a schema representing the lifecycle of a network. In that schema, for each kind of network problem, an informal probability distribution density describes how frequently that specific problem arise during the various phases of the lifecycle. Superimposed to this schema, the primary focus of the monitoring approach, of the testbed based tests and of the methodology described in this paper (TDND).

both in principles applied and in techniques and tools adopted.

The rest of this paper is organized as follows. Section 2 describes network problems and related risks which are addressed by current network testing practice. Section 3 reports the current practice in software testing. Section 4 briefly discusses the possibility of applying software testing methodologies to networking. Section 5 outlines the Test Driven Network Deployment methodology. Section 6 encompasses economic considerations and describes the tests used in TDND and the features which we impose to them to make the methodology affordable. Section 7 contains some examples of network test cases. Section 8 provides initial considerations about supporting tools for the methodology. Section 9 recalls some related work. Lastly, Section 10 provides conclusive remarks and intended future work.

2 Network Problems and Related Risk Treatment

Engineers design and deploy computer networks on the basis of a set of *requirements*. These requirements can be collected in several ways to express business goals and needs of the stakeholders at different levels of detail (see, for example, [43, 35]). However, a deployed network may actually not meet its requirements, due to *problems* in one or more of the complex steps involved in its lifecycle.

Each specific problem can be associated with the risk of having unmet requirements and economic losses. According to the relevance of a particular risk, it can be decided to accept or mitigate the risk. The most relevant risks should always be mitigated [49].

In the following we show which risks are considered by current practice in network testing and how they are mitigated.

2.1 Taxonomy of Network Problems

We do not believe that a taxonomy can be so sharp to provide a clear place for each network problem. However, in the explanation of our methodology, referring to a taxonomy is of great help.

A common way to classify network problems is to divide them in failures and human errors [30], as showed in the left side of Figure 1.

Failures are related to properties of network devices or of the environment surrounding the network; they comprehend acts of nature (e.g., earthquakes), hardware failures (e.g., link failure), and software failures (e.g., software crashes).

Human errors include all types of problems caused by actions performed by human operators. According to terminology defined in [45], human errors can be divided into slips and mistakes. *Slips* are oversight errors, for example, typos. *Mistakes*, on the contrary, are conceptual errors and can be further classified in *Analysis* and *Design* errors, according to the phase of the lifecycle in which they happen. A typical analysis error is the misunderstanding of the needs of the customer, leading to wrong or lacking requirements. Design errors encompasses both *Logical design errors* (e.g., wrong logical topology) and *Physical design errors* (e.g., wrong physical device choice).

Different network problems typically arise in different moments of the network lifecycle (see the right side of Figure 1). While the probability of having a failure evenly spans the operational phase, mistakes are related to analysis or design phases and slips to the deployment phase. This pattern is replicated in each change iteration during the operational phase. The methodology we propose in Section 5 exploits this regularity to perform deep tests when they provide the greatest value, that is, right after a change.

2.2 Current Practice In Network Testing

In current practice and methodologies [43, 36, 6, 46], network testing is mainly performed setting up test plants, also called testbeds, to validate the most critical design assumptions in the pre-deployment phases, and relying on monitoring tools to regularly check networks during operation. The primary focus and the risks mitigated by the approaches grounding on testbeds and monitoring tools are displayed in Figure 1.

In-depth testing is performed only in the pre-deployment phases on testbeds, which are a partial implementations of the final network. Testbeds do not catch the complexity of the production network and can only partially reproduce its behavior. Testbeds are suitable for mitigating the risks related to both physical and logical design errors, validating the most important design choices in a simplified environment.

After the execution of tests on testbeds, the network is deployed. Deployment is, generally, not subject to a deep testing activity; rather, network administrators manually perform scope-limited checks, based on the use of elementary debugging tools (e.g., traceroute). So, the network is almost blindly deployed hoping that, if prototypes work well, also production network will do.

Verification of the expected behavior of a network during operation is normally performed by a lightweight activity of monitoring. Monitoring is typically supported by specific tools either commercial (for example, IBM Tivoli NetView®, and HP OpenView®) or freely available (for example, Nagios®), which execute periodical solicitations of the most important machines of the network in order to verify their state and collect data for management activities. Monitoring aims at being not intrusive and, generally, does not encompass a comprehensive check of all kinds of requirements (see, for example, [50]). This approach focus on the detection of failures, but it is marginally suitable for detection of human errors and marginally mitigate the risks related to them.

In our opinion, none of the currently adopted testing techniques allows operators and administrators to in-depth verify production networks, supporting timely detection of the human errors that are added during the evolution of the networks themselves. Risks implied by human errors, arising in the deployment phases and during the change iterations, are mostly accepted in current practice, even if they are common, not avoidable and responsible for a great number of widespread disruptions and misbehaviors [4, 30, 15, 33, 3, 41, 40, 29].

3 Current Practice In Software Testing

Software engineers normally adopt an approach which is entirely different from common networking practice. They exploit testing to put much more attention on checking correct implementation and controlling side effects of every types of modifications, especially manually performed ones. Appropriate sets of comprehensive tests are often run for timely detecting human errors introduced during changes of software code. As a result, the testing activity is normally performed in-depth, in an automated way, using well-defined procedures and convenient tools [9, 12].

Among change-supporting software methodologies, Extreme Programming (XP) [11, 10] is one of the most appreciated. To facilitate the evolution of software and have guarantees, up to a certain degree, about correct implementation of requirements, XP chooses a peculiar approach to develop software, called Test-Driven Development (TDD) [8], in which testing has a primary role. In TDD, tests can be considered

as a formalization of the requirements collected in the analysis phase and both design and implementation of the code starts from the definition of a test. Each test is written before the implementation of the code, in a practice called test-first development, in order to encourage right definition of requirements and to set concrete goals for software production. This narrows the number of problems related to errors made during the analysis phase as well as in the translation of requirements into code; it also permits to define the moment in which it is possible to stop to program as the moment in which all tests are passed. Then, tests are considered as an asset and are maintained as well as the "application" software, so that they can be run every time a change is made on the code and can be frequently used to point out side effects of changes and to prevent error propagation.

According to TDD, software is developed in short iterations, each of which consists in the following steps.

1. Addition of a new test or modification of an existing one, to verify if the software accommodate a changed or an added requirement.
2. Sanity check, that is the check of failure of the new test just added as expected since the requirement the test verifies has not been implemented yet. Sanity check is a verification that the new test effectively works and catch an error on the unmodified software.
3. Implementation or modification of the code, aiming at fulfilling the new requirement.
4. Run of all tests ever written for the software application, in order to assure that changes made on the code effectively allow to meet the new requirement and, at the same time, do not affect previously implemented ones.
5. Refactoring of the code to achieve higher code quality.
6. Execution of all tests again.

Test-driven development normally exploits unit testing frameworks [9, 12], which are standard tools for creation and execution of automatic and repeatable tests. Those *unit tests* [1] are responsible for the verification of code units, which are single functions or the smallest sets of functions that can be verified separately from others. Tests of code units are made through the use of *test cases*: each test case represents an automated procedure able to check a single requirement the code unit has to accommodate. To easily run large sets of tests, test cases can be hierarchically aggregated in test suites, which are run by users as they are single tests.

Several experiences have demonstrated the effectiveness of TDD approach and related XP practices and positive results are achieved through the adoption, in the entire software lifecycle, of their refined testing techniques [34, 20, 37, 27, 24, 48, 31].

4 Can We Learn from Software Engineering Experience?

As well as software, computer networks are continuously evolving systems. We argue that computer networks can be practically treated like a software for several aspects, the most important of which we list below.

- Device configurations are the most relevant software-like part in a network and network operators type of work is somehow similar to that of software developers. They both work on text files written in formal languages, can quickly try changes and easily revert back, and usually are pressed to implement new requirements while keeping all the other features still working.
- The environment in which software and network engineers operate is typically heterogeneous, requiring the integration of products and components from different vendors and third parties. This type of environment also presents well-known problems in maintenance and evolution of products, like those related to the update of some components of the system. For example, in software integration business, developers rely on third party software, which is subject to frequent updates, that can impair also a previously working system. In networking, the firmware of the devices is analogously maintained by third parties and is subject to updates, posing problems that software engineering are used to face.

- The part of the network topology that does not directly rely on wiring may be quickly and cheaply changed, as software code, for its nature, can be. Note that many technologies and protocols, like MPLS, VLAN or tunnelling, virtualize hardware resources and tend to make the topology easier and less expensive to change.

- Software developers can perform periodical clean-up of the code without altering the implemented functionalities, in a practice called refactoring [22, 39], performed to keep high the quality and lower the complexity of the code itself. In networking, analogous activities, like, for example, IP address renumbering, configuration clean-up, network and autonomous system merging, are usually performed with the same purposes.

- Both software and network engineers have to cope with unexpected side effects of changes, since unpredictable consequences can easily arise in inherent complex systems, like computer networks or software systems. Side effects of changes can also be widespread and latent [26, 44] and can provoke mismatchings with requirements and disruptions on the entire system.

We think that these similarities between software and networks enable to apply approaches and solutions used in one of them to the other, especially for analogous problems. We believe that the software testing practice can stimulate a more mature use of testing in networking. We argue that a software-like network testing activity will make network administrators much more confident in modifications they have to perform, decreasing the number of disruptions due to unforeseen side effects of changes and to unpredicted and inevitable human errors. The proposed approach is also expected to enable further possibilities to enhance the quality of continuously evolving networks, better supporting activities similar to software refactoring.

Obviously, these approaches can not be directly applied to computer networks, firstly because of the different nature of networks with respect to software. Consider that software is immaterial (do not physically break or deteriorate) while networks mainly consist of cables, computers and devices. Main difficulties in making software testing approaches suitable for networking are relative to the following aspects.

- The behavior of a software program is verified, in TDD, using other software, expressly developed for testing activity. On the contrary, network tests can not rely only on network technologies, since network protocols, standards and devices are focused on communication and are not appropriate to support testing activity. We argue that network tests have to be realized as software programs, both for easing their execution and for making them affordable and effective. Sections 5 and 6 contain more reasoning on this topic.

- Being immaterial, software allows to easily and cheaply create fictitious and conveniently rearranged realistic environments which can be specifically used for testing. On the contrary, in networking, setting up a test environment, with the same features and complexity of the production network, is normally impractical.

  As described in Sections 5 and 6, our methodology is conceived to be directly applied to operational networks. However, since tests performed on an operational network can hurt the network itself, provoking disruptions and misfunctionings, we pose great attention on dealing with intrusiveness of tests, both from a methodological point of view (see Section 5) and at the level of single test (see Section 6.3.2). In perspective, we think that it might be possible that virtualization techniques [7, 18, 42] can further support our testing approach, allowing operators to conveniently set up ad-hoc virtual environments that closely mimic the real network. In this case, TDND will gain great benefits from the adoption of virtualization techniques, since tests used in the methodology can be more affordable, appliable and easier to be run.

- While in software engineering the analysis activity normally relies on the production of well-defined documents, like, for example, use cases, story boards and contracts of operations, literature in networking field does not encompass standard languages and techniques for formalizing all types of different requirements. Current methodologies [43, 28, 51, 35] propose different semi-formal ways for collecting and expressing requirements at several levels of abstraction; moreover, many works, like [23, 21, 17, 52] for QoS requirements, propose their own formal methods and languages. However, none of them is appliable to all kinds of requirements and is recognized as a standard. This can be partly considered as a consequence of the intrinsic high coupling of computer networks.

As a result, the definition of the scope of a network test (e.g., the requirement it verifies) may be not a trivial activity. We propose to use self-describing tests as formalizations of the network requirements they verify (see Section 6.2).

Sections 5 and 6 present further details on the solutions we propose for addressing the above difficulties in adapting software techniques to computer networks. In Section 5, we describe the main principles of the Test Driven Network Deployment methodology, which is inspired by successful approaches used in software, namely Test Driven Development, while Section 6 details our proposal for adapting software unit testing to be suitable and affordable in networking domain.

5 The Test-Driven Network Deployment Methodology

The main prescription of the TDND methodology is to use automatic testing to timely give evidence of correct or incorrect implementation of analysis and design choices. As in TDD, the main idea is that large sets of tests should be run, with negligible costs, right after each change is deployed on the network, in order to deeply verify potential human errors and discover side effects of changes.

We call network test cases the tests used in TDND and we define them as follows.

A *network test case* is a sequence of operations that can be performed to verify the actual behavior of the network, and, by extension, the software that implements those operations (taking care of configuration, execution and evaluation of the test). The result of a test is either *success* or *failure*, according to the comparison of the behavior detected with the required one.

As in TDD, the success of a network test case is of great value, since it provides a certain level of confidence about the correct implementation of the requirement it verifies. Also, a failure is valuable since, in TDND, it provides a timely detection of a problem in a production network.

In our approach, the team that deploys the network, also builds, maintains and extensively runs a set of network test cases, which we call *Project Test Set*. This set of tests is considered an asset of the project, as well as the network itself.

The overall process of introducing network test cases in a project brings new costs and new advantages. A deeper discussion on economic considerations and affordability of TDND is provided in Section 6.

Now, we introduce TDND in a simplified form with the purpose of highlighting the main principles it relies on. The full methodology with all technical aspects is presented later in this section.

TDND is based on the organization of networking activities in short iterations, each consisting in the execution of the following major steps, which are directly inspired by TDD.

1. **Formalization.** Formalize a new network requirement as a new network test case or a modification of an old requirement as a modification of an existing test case. Requirements can be elicited in the analysis phase or can arise from a troubleshooting activity.

2. **SanityCheck.** Check the failure of the network test case just added, as expected since the requirement the test verifies has not been implemented yet. This step ensures the effectiveness of the new network test case.

3. **Deployment.** Modify the network in order to pass the new network test case.

4. **Testing.** Run all the test cases for the network, in order to assure, with a certain degree of confidence, that the changes made effectively allow to meet the new requirement and, at the same time, do not compromise previously implemented ones.

In this form, TDND does not address the intrusiveness of tests, which we consider a very relevant aspect in network testing. To formally define the intrusiveness of a test we first need to introduce the concept of status of a network.

We call *status of the network* the collection of all the parameters which affect the behavior of the devices with respect to user traffic. Configurations are obviously part of the status of the network but even more low level parameters belong to it (e.g., interface status, routing protocol status, VRRP status).

In the following, we always make a *stationarity assumption*: we assume that the status of the network (and hence its behavior) changes only if an operator or a network test case deliberately modifies it. For example, we exclude that situations like unexpected link failures happen during the execution of tests.

type of test	**network status**	**examples of typical operations**	**possible goals**	**expected amount per project**
safe	not changed	inject well formed expected traffic	test routing	really many
		inject well formed unexpected traffic	test security policy	many
intrusive	potentially changed	inject malformed or malicious traffic	test for firmware bugs	some
	intentionally changed	temporarily change configuration	test fault tolerance	some

Figure 2: Proposed classification of the network tests according to their intrusiveness level.

We define the *intrusiveness* of a network test case as its capability of changing the status of the network and, possibly, the external behavior of the network itself, in terms of functionalities offered to the users.

In Figure 2, we propose a classification of the network test cases according to their intrusiveness level. This classification is used in our methodology for deciding how and when tests should be run. The first category of network test cases is composed by procedures which intend to check how the network behaves in its current state. These tests do not change the status of the network and can be thought as a sort of "read-only" tests. In this sense, they are *safe*, since they are unlikely to cause any type of disruption. The second category comprehends network test cases which do intend to change the status of the network, either deliberately (e.g., for testing failover configurations) or to simulate the behavior of a malicious user (e.g., to test for a possible firmware bug). Tests of this kind are called *intrusive* and the risks related to their execution are tangible.

If all the network test cases are safe, TDND can be applied as described above. However, in the most complex contexts, some requirements can only be verified running intrusive network test cases. In perspective, we think it will possible to eliminate the risks related to intrusive tests running them on virtual environments [7, 18, 42] that closely mimic the real network. Following a more conventional approach, the mitigation of the risks related to intrusive tests can be done by running them in appropriate time slots (e.g., nightly or on the weekend) and/or using safety measures (e.g., presence of a network administrator). For this reason, each test case should be annotated with its *intrusiveness level*. Details about annotation of network test cases are provided in Section 6.

The full TDND methodology, that handles intrusive network test cases, prescribes short iterations, each consisting in the following major steps.

1. **Formalization.** Formalize a new network requirement as a new network test case or a modification of an old requirement as a modification of an existing test case. Requirements can be elicited in the analysis phase or can arise from a troubleshooting activity.

2. **Intrusiveness Assessment.** Assess the intrusiveness level inherent to the network test case just added and annotate the new test case accordingly. Add the new network test case to the Project Test Set.

3. **Conception.** Decide which changes will be implemented to meet the new requirement, and select the group of network test cases you want to run immediately after the deployment of modifications, in order to check for the most likely side effects of the changes. In the most common situations, selecting all safe test cases should be enough, however, intrusive test cases may be needed in special circumstances.

4. **Planning.** Schedule next steps of the iteration according to the risks related to the changes to be deployed and to the assessed intrusiveness level of the network test cases just selected. In this phase the following rules should be observed for the scheduling.

    - The test cases selected in the **Conception** phase should be run as soon as possible after the implementation of the change.

- If some of the selected tests are intrusive, all the following steps of the iteration should be planned to be executed in appropriate time slots and using cautionary measures.
- The full Project Test Set should be executed as soon as possible after the deployment.

5. **SanityCheck.** Check the failure of the network test case just added, as expected since the requirement the test verifies has not been implemented yet. This step ensures the effectiveness of the new network test case.
6. **Deployment.** Modify the network in order to pass the new network test case.
7. **Testing.** Run the network test cases decided in the **Conception** phase. Run again all the network test cases in the Project Test Set, as soon as possible, as scheduled in the **Planning** phase, in order to assure, with a certain degree of confidence, that the changes performed on the production network allow to meet the new requirement and, at the same time, do not compromise previously implemented ones.

6 Affordable Testing

Ideally, TDND permits to deeply verify, at any moment, all the network requirements, at an arbitrary level of depth and coverage. However, we think that, to keep the project remunerative, only a small part of the budget can be typically dedicated to testing. In this section, we aim to show that the adoption of TDND can be profitable regardless of the budget dedicated to it, with benefits that increase along with the effort. This makes TDND also suitable to be incrementally adopted in already active projects, whose budget is mainly dedicated to maintenance.

We believe that initial costs for adopting TDND (mainly training of the personnel) can be greatly limited by the use of intuitive supporting tools, that we plan to develop as a follow-up of this paper. Operational costs encompass development, run and maintenance of tests. Both initial investments and development costs are quickly amortized because of the frequent use of the tests. Further, to lower all the operational costs, we aim at making network testing activity effective, easy to perform, repeatable, and highly automated, as it is in software engineering. We also suggest a selective code sharing approach which is peculiar of TDND. To achieve these *objectives*, we mandate the network test cases to have certain features. In the following we better describe the objectives for the testing activity and the relationship with affordability. Later in this section, we also detail each of the features we impose to network test cases. Some of these features ask for appropriate supporting tools, whose design and main functionalities we outline in Section 8.

6.1 Objectives for Network Testing

We identify the following primary objectives for the testing activity.

- **Effectiveness.** Testing activity must be capable to correctly verify the behavior of the network. Effectiveness of tests encompasses coverage and depth of tests performed.
- **Ease.** Network operators must be able to execute tests in a simple and straightforward way and to understand their results at a glance.
- **Repeatability.** Tests must be always ready to be executed, so that it should be possible to repeat groups of tests every time a network operator needs it.

These objectives are strictly related both to automation and affordability. Effectiveness of testing activity is proportional to the number of requirements actually checked and to the type of inspections performed by tests. For this reason, effectiveness of testing can be increased by increasing the costs, using a greater number of tests to perform deeper inspections, but this approach is strongly limited by budget considerations. However, in TDND, the Project Test Set is an asset and even a small effort can lead, over time, to large effectiveness. Moreover, ease to perform and repeatability combine to make the testing activity more automated and less expensive.

Notice that efficiency is not accounted as a primary goal of testing. Since efficiency largely depends on individual test cases and on their implementation, assurance of efficiency is mainly delegated to the developers of network test cases. Aspects relative to the efficiency of network testing are further discussed in Section 6.4.

6.2 Mandatory Features of the Network Test Cases

To achieve the objectives for network testing, a network test case is characterized by a set of well-defined features. In the rest of the paper, we always assume that all the network test cases have the features introduced in the following.

Conceptually, network test cases play the same role of software test cases in TDD [12] and inherit from them the following features.

- **Requirement oriented.** A network test case is targeted to the verification of a single network requirement and addresses the problem of checking if the network conforms to it. Hence, a network test case can be considered as a formalization of the requirement it verifies.
- **Boolean.** The result of the execution of a network test case is either "success" or "failure".
- **Autonomous.** Network test cases are autonomous, in the sense that they can be run by a human operator without any input data to be interactively specified.
- **Stateless.** Network test cases do not maintain an internal state across different executions. This implies that the results of each run of a network test case are deterministic under the stationary assumption, since they always produce the same results if the status of the network is not changed.
- **Composable.** A *network test suite* is a set of network test cases grouped with the purpose to be collectively identified and executed. A network test suite can be treated by the user as a single network test case and it is successful when all the test cases included in it are successful, otherwise it fails. The Project Test Set and the sets of tests created during the conception phase of the TDND methodology (see Section 5) are examples of network test suites.

Not all the features of software test cases are appropriate for network tests. We now present the distinguishing features of the network test cases.

- **Explicitly dependent on each other.** Since a network requirement normally relies on the implementation of other requirements (e.g. a test of TCP based services can succeeds only if IP level connectivity properly works), the result of a network test case generally depends on the result of other network test cases. The dependencies between tests should be explicitly managed, so that groups of network test cases can be easily run respecting their dependencies. This approach allows to verify the requirements in a correct order and to univocally attribute the result of a test to the requirement it verifies, since all the requirements, on which the test depends, can be verified before the test itself.
  Discussion. In software testing it is possible to perform pure unit testing, separately verifying each function as a unit, so that errors in a function does not affect tests of other functions. This is enabled by the implementation of modularity and low coupling principles and is achieved using techniques like "mock objects" [32]. Problems related to the interaction among functions are discovered by "integration tests" [13]. Unfortunately, at the moment, the technology does not allow to adopt a similar approach in networking, since it is not easy to realistically emulate a part of a network. As a consequence of the high coupling of computer networks, network testing always have to deal with many parts of the operational network at the same time and tests can be hardly made independent from each other. Notice that the dependencies between network test cases implies a partial order relationship between them and the order of execution of the tests in a network test suite can be automatically computed through a topological sorting.
- **Annotated and Selectable.** Each network test case is characterized through a set of annotations whose semantic should be made clear to operators. Annotations convey relevant information on the network test case they refer to and are used both for automatic selection of network test suites in a flexible way and for documentation purposes, so that further written documentation is not needed. Some annotations, like for example those related to the requirement verified, the intrusiveness level and the dependencies of the test, should be mandatory, in the sense that they should be attached to every network test case. Annotations can be implemented as key-value pairs.
  Discussion. In software engineering, there exists a natural way to aggregate test cases in test suites according to the hierarchy induced by the decomposition of software in modules. On the contrary, we think that there is not a straightforward hierarchical organization for network tests and criteria

for the aggregation of network test cases can depend on the specific situation. For example, tests to be run immediately after a change (see Section 5) can be selected according to the project, to the level of the protocol stack involved, to the physical part of the network to be tested, to the kinds of users it impacts, to the intrusiveness level, or all these criteria can be applied at the same time. For this reason, we propose to create network test suites grounding on an annotation based selection of the test cases. Annotations also allow to attach documention to tests (see Section 7 for further details).

The table presented in Figure 3 shows how the desired objectives for the testing activity are achieved by means of the features of the network test cases.

6.3 Optional Features of the Network Test Cases

In this section, we discuss some optional features of the network test cases, which we think are relevant for assuring the affordability of the network testing activity. These optional features are intended to be alternatives between two options, but we mandate all the network test cases to be characterized by one of these two options.

6.3.1 Granularity of Network Test Cases

We roughly divide network test cases in fine-grained and coarse-grained.

- **Fine-grained** network test cases are those which involve one or few operations and are typically focused on a single device. It is frequent that this type of tests consider the network or a consistent part of it as a black box. Fined-grained tests are similar to the simple checks performed by monitoring tools.

- **Coarse-grained** network test cases involve more complex examinations which normally comprehend more than one device. The approach followed in these tests is similar to gray box tests.

Note that the possibility of being coarse-grained is a discriminating feature of the network tests with respect to software ones, since current unit testing practice in software engineering prescribes to have only fine-grained test cases. This is possible since, in software engineering, keeping coupling low is one of the most important principles. Low coupling principle greatly eases management and evolvability, and induces the definition of cohesive software modules which can be checked by simple separated tests. On the contrary, coupling in networks is inherently high. This makes network testing more challenging than software testing, since it is hard to decompose a network in subsystems with clear and well defined behaviors. However, the coarse-grained feature can be exploited to guarantee relevant properties of tests, like safety or enhanced efficiency, as described in Sections 6.3.2 and 6.4.

The table in Figure 4 displays how both fine-grained and coarse-grained network test cases help to accommodate most of the primary objectives for network testing.

However, the table also shows that, while fine-grained tests are easy to be created and managed, coarse-grained network test cases can be difficult to realize and expensive to maintain. For this reason, we propose to arrange community shared libraries of coarse-grained network test cases.

- **Community shared.** More common and complex network test cases should be collected in libraries, so that they can be applied to different networks with minor configurations. Test libraries should be publicly available, so that network operators can choose and pick tests from them when necessary.
  Discussion. In software testing, this type of support can not be provided since the functionalities of a specific software are, normally, very peculiar to that software. On the contrary, the purposes of computer networks are quite homogeneous, both for requirements and implementation, since networks are specifically oriented to communication. This feature can be profitably exploited providing appropriate libraries which cover the testing of a large number of common requirements. The network test cases belonging to these libraries are likely to be quite complex and, hence, should be conveniently annotated to simplify their use.

Tests from the community shared libraries may need an initial configuration, in order to adapt to specific networks. If the testing software need to be configured (for example providing the IP addresses of

	Effectiveness	**Ease**	**Repeatability**
Requirement oriented	Each test verifies the correct implementation of a single network requirement.		
Boolean		Results are clearly understandable and can be easily aggregated.	
Autonomous		Running a test requires negligible effort.	No human interaction is needed.
Stateless	Results only depends on the status of the network.		Results only depends on the status of the network.
Composable		Arbitrary large sets of tests can be run as if they were one.	The same selection of tests can be run many times.
Explicitly dependent on each other	The result of a test can be univocally attributed to the requisite it verifies.	Proper execution order for tests can be automatically computed.	Proper execution order for tests can be automatically computed.
Annotated and selectable	Annotations allow to select proper test set for the situation.	Annotation-based selection can be easily performed.	The same selection can be repeated over time.

Figure 3: The table shows the aims achieved through every feature of the network test cases.

	Effectiveness	**Ease**	**Repeatability**
Fine-grained	Fine-grained tests are appropriate for simpler requirements.	Simple to verify requirements can be straightforwardly checked with tests which are easy to write.	Fine-grained tests are formalized and always available to be run.
Coarse-grained	Coarse-grained tests consider all aspects and elements affected by complex requirements.		Operations for complex tests are always performed in the same way.

Figure 4: The table shows how both fine-grained and coarse-grained features help to make the testing activity more effective and repeatable. Note that coarse-grained tests do not satisfy the ease objective, but makes testing activity potentially more difficult and expensive.

	Effectiveness	**Ease**	**Repeatability**
Community shared coarse-grained network test cases	Coarse-grained tests consider all aspects and elements affected by complex requirements. Contributes from many people increase effectiveness.	Ready to use coarse-grained tests are publicly available.	Operations for complex tests are always performed in the same way.

Figure 5: Community sharing makes testing activity easier and less expensive also for coarse-grained tests.

the machines to be checked), the configuration is considered part of the network test case. Developers of the shared network test cases should aim at minimizing the effort of the user in writing and managing the configuration of the tests. We propose to automatize some of these activities (like versioning and recovery of the configurations of the tests), relying on convenient functionalities of appropriate supporting tools, as described in Section 8. Anyway, we estimate that the cost for the configuration activity is normally much less of the cost of writing a complex test from scratch.

Figure 5 highlights the benefits of sharing coarse-grained network test cases within a community.

We expect that both fine-grained and coarse-grained network test cases are used by an organization on the same projects. Privately maintained fine-grained network test cases are adopted for performing simple checks (e.g., verification of availability of a specific Web server) which are very peculiar to the specific project and they can be similar to solicitations performed by monitoring tools. On the contrary, coarse-grained network test cases are selected from publicly available libraries to implement more complex but common examinations (e.g., IP level global connectivity).

	Effectiveness	Ease	Repeatability
Safe		Safe tests do not need to define restoring procedures.	The status of the network is not changed by testing activity.
Recoverable		Restoration of the initial network state, when needed, is automated as much as possible.	In the worst case, a semi-automatic procedure to restore the initial status of the network is given.

Figure 6: Safe and recoverable network test cases helps to achieve the objectives of the network testing.

6.3.2 Safety

Testing must not permanently hurt the network. In particular, each execution of a network test case should leave the network in the same state in which it was before the run of the test.

From a methodological point of view, TDND prescribes to dispose appropriate countermeasures for intrusive network test cases (see Section 5), which, by definition, do not give guarantee about the final status of the network. However, we also mandate that network test cases can exclusively be either safe or recoverable.

- **Safe.** Network test cases which do not change the status of the network can not hurt the network by definition.
- **Recoverable.** Intrusive tests should be equipped with an automatic, or at least a semi-automatic, restoring procedure. Automatic restoring procedures should always be executed as the final step of each run of a network test case. If the test case crashes or terminates without being able to restore the initial status of the network, it should return a failure and it should launch the restoring procedure, at least proposing to the user a semi-automatic way to restore initial state of the network. It is also convenient that restoring procedures can be executed independently from the network test cases they are attached to.

Figure 6 displays how both safe and recoverable features of the network test cases are appropriate for the objectives of network testing declared in Section 6.1

Details on how to create restoring procedures which are both effective and automatic are mainly delegated to developers. Supporting tools presented in Section 8 should be designed for facilitating the work of developers in the definition of the recovering procedures. However, we expect that the network test cases which have to deal with safety are mainly the coarse-grained ones, and, hence, community shared.

Notice that software test cases do not need to deal with safety since, in the software field, the testing environment is not a production one, but it is created only for testing purposes. In TDND this is not true.

6.4 Efficiency and Tradeoffs

In Sections 6.1 and 6.2 we have not considered the efficiency as a primary objective for the testing activity neither as a feature of the network test cases. We argue that, within certain limits, TDND can be profitable also with inefficient network test cases. Nevertheless, we know that inefficiency might negatively affect both effectiveness and affordability of testing activity, since inefficient tests are normally quite impractical and expensive and cannot be frequently executed, forcing operators to postpone the moment in which to run them. Moreover, the more efficient the tests are, the more realistic is the

(stationarity) assumption that the network is stable during their run and the less is the effort needed for the planning activity.

Enhancing the efficiency of the network test cases is an important task which is mainly delegated to test developers. In the following we present some general considerations about efficiency and about the way we think to help test developers in dealing with it.

The efficiency of a network test case is primarily related to its computational complexity. Namely, a test case should run in short time, produce low traffic level, and have low memory occupation. We expect that, in the large majority of the test cases, memory occupation will not be a problem and traffic generated by tests will be largely acceptable considering current bandwidth and network performance guarantees. On the contrary, we think that running time may be critical, due to factors like latency, and test developers should focus on it.

Efficiency problems can be generally tackled by making appropriate assumptions on the properties of the network to be verified. We define assumptions of a network test case as the hypothesis which affect its execution but are not verified before a run. Assumptions generally encompass hypothesis on technologies, protocols, and tools available for the test. Making more or more relevant assumptions can allow to reduce the algorithmic complexity of a network test case, but can also decrease the applicability of the test.

Moreover, efficiency and assumptions are both affected by the effort required for the configuration of the tests, in the sense that requiring the user for more configuration effort can allow to decrease the number and the relevance of the assumptions made and lower the algorithmic complexity of the test. However, we think that test developers should normally aim at minimizing it for easing the applicability of the TDND methodology. We expect that the configuration effort can be relevant only for coarse-grained tests, which are not specific of the network and need to be configured (see Section 6.3). Nevertheless, also notice that the coarse-grained feature of the more complex network test cases allows to make some optimization, for example, on the number and the type of operations executed by the software program and data structures used in the tests. Investigation of optimized procedures and algorithms, with the intent to limit the complexity of the tests, along with the amount of traffic generated, is marginally encompassed by this paper in Section 7.3 and can be an interesting subject of further research.

Moreover, notice that only a tradeoff between assumptions, configuration effort and efficiency of the test can be practically reached. In order to enable network operators to choose the most suitable tradeoff for their needs, it is useful to have (e.g. in the community shared libraries) different network test cases verifying the same requirement with different tradeoffs.

For this reason, we think that, for each network test case, annotations about both its efficiency and the assumptions on which it relies are highly recommended.

Annotations about dependencies and assumptions also allow to automatically optimize the execution of groups of network test cases, relying on appropriate functionalities of the supporting tools we plan to realize (see Section 8 for further details). For example, tools are enabled by these annotations to conveniently make the execution of the tests parallel, maximizing the throughput and respecting dependencies of tests at the same time.

7 Concrete Examples of Network Test Cases

In this section, we provide some practical examples of test cases, we show those that according to us should be mandatory annotations, and we propose some optional annotations. We suggest to use optional annotations for documenting network test cases, especially coarse-grained ones. We also present in the following two concrete examples of network test cases.

7.1 Structure

According to all the considerations discussed so far, we propose the following structure for test cases.

- A **core** part which contains the smallest possible set of information needed to run the test, containing
  - the *procedure*, that is, the software code which implements the test, and
  - the *configuration* that is the data which are needed by the procedure to run on a specific network.

- Each network test case must be equipped with **Mandatory Annotations**, which are annotations that all tests should have since the information stored in them will be strictly needed by the supporting tool we envision.
  - *Name.* An expressive name for the network test case, which identifies it.
  - *Requirement.* The identifier of the requirement verified by the network test case.
  - *Dependencies.* The identifiers of the requirements on which the network test case depends. It is intended that the tests which verify these requirements should be run before this one.
  - *Intrusiveness.* The assessed intrusiveness level of the network test case.
  - *Restoring Procedure.* The automatic or semi-automatic restoring procedure. Only for intrusive tests. This could have a complex structure encompassing, for example, a description, something that should be "run" before and/or after the test, configurations to be collected, etc.
- Each network test case may be equipped with **Optional Annotations**. We expect them to be especially useful for coarse-grained and community shared tests. The following are some we think can be highly recommended.
  - *Objectives.* The description of the objectives of the network test case.
  - *Assumptions.* The assumptions made by the network test case on the capability of the network.
  - *Algorithm.* A description of the algorithm implemented by the network test case.
  - *Complexity.* About the algorithmic complexity of the network test case concerning time and generated traffic.
  - *Configuration Guidelines.* Instruction on how to configure the test.

7.2 Examples

In this subsection, we present two concrete network test cases. The first example is a fine-grained test. The second test case is a coarse-grained one, which can be considered as a candidate community shared test, and it is described specifying a series of conveniently chosen optional annotations. Notice that, in both examples, network test cases are largely self-describing and need no further documentation.

7.2.1 Example 1: Gateway Reachability TestCase

Name. Gateway Reachability TestCase.

Requirement. Configured default gateway.

Dependencies. None.

Intrusiveness. Safe.

Restoring Procedure. None.

Description of Test Objective. The test verifies if a specific host can reach its default gateway to communicate with machines and devices which are external to its LAN.

Assumptions.

- Physical connectivity.
- The tester can remotely access through the SSH protocol (using the public key authentication mode) to the host whose connectivity to its default gateway must be checked.
- The ping command is effective on the LAN considered.

Complexity. Constant time. Constant generated traffic.

Procedure. *< code to be executed >*

Configuration. *< address of the machine subject to this test >*

7.2.2 Example 2: Inter LAN-Cluster TestCase

Name. Inter LAN-Cluster TestCase.

Requirement. Inter-LAN IP connectivity.

Dependencies.

- Intra-LAN IP connectivity.
- Configured default gateway.

Intrusiveness. Safe.

Restoring Procedure. None.

Description of Test Objective. The test verifies whether every host is able to send and receive IP packets to hosts attached to different LANs of the same enterprise network.

Assumptions.

- Physical connectivity.
- Connectivity is a symmetrical relationship: if a host A can ping another host B, also the vice versa is assured.
- Connectivity between hosts of the same LAN is guaranteed.
- Routers which are default gateways for at least one host do not filter packets and traffic flows.
- The tester can remotely connect to all the LAN representatives through the SSH protocol (using the public key authentication mode).
- The traceroute probes are not blocked by routers of the enterprise network.

Algorithm. Each LAN is represented by a machine, as configured by the user. In this algorithm, pings among LANs are performed by pinging representatives of the LANs. The algorithm checks the connectivity of the tester machine to every LAN representative. This series of checks is performed using the traceroute command, in order to also collect some information about the topology of the network. This information is used to infer the default gateway of each LAN, which in turn allows the identification of the LAN clusters. A *LAN cluster* is defined as the maximal set of LANs with the same default gateway. The set of LAN clusters is a partition of the set of the LANs. For each pair of LAN clusters (A, B), the algorithm executes a minimum number of pings so that all l_A LANs of A are start of at least a ping and all l_B LANs of B are destination of at least a ping. The algorithm performs $\max(l_A, l_B)$ pings for each pair of LAN clusters.

Complexity. Say c the number of LAN clusters and l the number of LANs, the algorithm performs $O(cl)$ pings.

Procedure. *< code to be executed >*

Configuration of the Test Case. *<In the configuration of the TestCase, the IP address of one and only one host have to be specified for each LAN. The host specified for each LAN gains the role of LAN representative. Connectivity is practically checked only between LAN representative, since the connectivity within each LAN is guaranteed by the assumptions made. >*

7.3 An Example Catalogue of Coarse-Grained Test Algorithms

We provide an example catalogue of coarse-grained tests appliable within the TDND methodology. The testing procedures we propose aim at verifying basic functional requirements, like IP-level connectivity among LANs or within a LAN. This catalog is not at all exhaustive. Also, we think that algorithms sketched, as well as related testing problems, can be subject of further research. All the procedures are informally showed, describing the main idea at their basis, and very briefly discussed.

We present algorithms grouped according to the requirement they verify. More than one procedure is proposed for some requirements, since they realize different tradeoffs (as already discussed in Section 6.4). A comparison between the different procedures is reported in Figure 7.

Notice that, each procedure sketched in this section does not specify details about concurrency solutions (e.g., for enhancing performance), low level primitives (for example, hping, tcptraceroute or nmap for sending TCP packets) and protocols (ssh, telnet or similar to remotely access network devices), etc.

7.3.1 Inter-LAN IP Connectivity

The procedures listed below are targeted to verifying if every host is able to send and receive IP packets to hosts attached to different LANs of the same enterprise network. We assume that the set of LANs to be verified are an input of the algorithm (for example, in the configuration of the test case).

Brute Force Host Ping. The simplest procedure to check inter-LAN IP connectivity is to verify, for each possible pair of hosts belonging to different LANs, that they can ping each other. This approach is practically infeasible, since the number of pings required is a quadratic function of the number of hosts of the network .

Brute Force LAN Ping. This procedure is essentially an optimization of Brute Force Host Ping. Assuming that intra-LAN connectivity is guaranteed, a host can be used as the representative of its LAN. Hence, exhaustive tests between all possible pairs of representatives of different LANs are performed.

Inter LAN Cluster Procedure. For each LAN, it is chosen a host which represents all the hosts of that LAN. Moreover, the sets of LANs which share the same default gateway (LAN clusters) are identified, either using traceroute commands or user configuration settings. Ping is then used to check connectivity between LAN representatives of different LAN clusters, further reducing the number of pings with respect to Brute Force LAN Ping algorithm. For more details on this procedure see Section 7.2.2.

Loose Source Routing Procedure. This procedure exploits the loose source routing mechanism, to impose routers that must manage the packets used in the test. Hence, injecting a convenient group of packets allows to verify the correct configuration of all routers and the reachability of hosts placed on different LANs from any point of the enterprise network. Unfortunately, enabling loose source routing on devices is commonly deprecated for enhancing security and, for this reason, this procedure may be impractical.

7.3.2 IP Filtering

The IP Filtering requirement is relative to the verification of the filtering policies implemented by routers and stateless firewalls. In particular, the purpose of the following procedure is to check that pairs of hosts which should not ping to each other can not effectively do it. Hosts to be checked are considered input to the algorithm proposed. Notice that, if filtering is made per LAN, the verification of unreachability between two hosts involves unreachability between corresponding LANs. The IP filtering requirement is somehow complementary to the previous inter-LAN IP connectivity one.

Probing Procedure. Given the pairs of hosts for which the traffic should be filtered, for each pair, a ping command is executed from one of the hosts of the pair to the other, expecting for the loss of the probe packets. Ping commands can be executed in parallel by different hosts. This procedure can be run by a single tester if it uses ssh or telnet to remotely access other machines or if source routing is enabled.

Procedure	Pros	Cons
Brute Force Host Ping	The procedure is easy to implement and to be made parallel.	Practically infeasible due to excessive number of checks (ping) to perform.
Brute Force LAN Ping	The number of ping is related to the number of LANs (not of hosts).	The assumption that a host can represent all the hosts of its LAN must be realistic (e.g., any filtering is made based on the LAN subnets).
Inter LAN Cluster Procedure	The procedure reduces the number of checks if the network contains LAN clusters.	The knowledge or, at least, the deduction of part of the topology of the network is necessary, affecting either the configuration effort or the efficiency of the test.
Loose Source Routing Procedure	The procedure can encompass optimization of checks (more devices can be verified using a single packet); remote connection is not needed.	Source routing is typically disabled on real networks, due to security issues.
Probing Procedure	The procedure is easy to implement, also in a parallel version.	The procedure is quite inefficient, since each pair of hosts requires a check.
Configuration Inspection	A minimal amount of traffic is generated and time complexity can be lowered.	The definition of proved patterns is needed, and no evident proofs of the actual behavior of the network are collected.
Experimental Check	Actual behavior of the operational network is verified dynamically.	This procedure is intrusive, and, hence, ask for cautionary measures.
Exhaustive Port Check	Procedure is easy to implement and to be made parallel.	The procedure can be inefficient, both for traffic generated and time needed.
Client Group Verification	A homogeneous group of clients can be verified minimizing the number of checks.	Knowledge of part of the topology and pre-computation are needed.

Figure 7: The table resumes the main features of the procedures presented in this paper.

7.3.3 Backup Link Configuration

Several networks have to guarantee fault tolerancy, at least at a certain degree. In this area, a common practice is to configure a link as a backup link, so that traffic flows through it when the primary link fails. The procedures listed in this section aims at verifying the correct configuration of backup links.

Configuration Inspection. This test checks the adherence of the configuration of devices to patterns which are already known to be properly working. The devices whose configurations are inspected, can be those either connected to primary or backup links. The test is safe, however, the identification of correct patterns can be a difficult activity and the set of pattern used in test can be not complete with respect of all correct configurations.

Experimental Check. The most reliable way to verify the reaction of a network to a physical failure is to provoke the failure in the operational network and check for the expected behavior. This means that a failure on the primary link is simulated (e.g., shutting down an interface of a device connected to that link) before testing the correct operation of the backup link. A similar test is clearly intrusive and potentially harmful. For this reason, the change of the status of the network must be adequately controlled by the implementation of the test, providing the user with an effective restoring procedure (see Sections 5 and 6.3.2).

7.3.4 Wide Availability of a Service

A possible requirement relative to the application level of the protocol stack is the availability of a remote service that can be accessed from a large set of hosts. From the point of view of the network administrators, we think that verifying availability of a service can mean check for the reachability of the open TCP port on which the service is running. We believe that deeper tests on the status of the services and on their responses to different types of requests can be done within TDND, but more information is needed on the specific service verified. The identifiers of both clients and server machine, the protocol and the port on which the service runs are considered input of the algorithms suggested below.

Exhaustive Port Check. The procedure consists in the exhaustive verification of the service availability from all the clients. For each client, the test verifies that the port of the service appears reachable and open to it.

Client Group Verification. The clients are classified in groups, according to the LAN on which they reside. Then, checks are performed from only one element (representative) of each group of client to the service to be verified. This procedure is potentially more efficient than the previously described exhaustive test.

8 Supporting Tools

In Section 6 we described some features we mandate for the network test cases. We also anticipated that many of those features naturally ask for automated support. In this section, we provide some initial considerations about supporting tools for TDND which we consider a crucial factor for the success of the methodology.

We think that supporting tools should, at least, meet the following functional requirements.

Storage. The tools should support the homogeneous storage of network test cases, both community shared and privately written ones. Possibly, the storage format should be open so that the asset represented by network test cases is not tightly linked with the tools.

Consistency. The tools should allow the user to check for consistency of the network test cases (e.g., for dependencies annotations).

Selection. The tools should allow the user to easily select the suites of test cases to run. Since there is no restriction on optional annotations, we expect the selection tools to be quite flexible and maybe borrowed from other storage technologies (e.g., SQL, XPath).

Execution. The tools should allow the user to easily and automatically execute large suites of network test cases. The sequence of the execution of the tests should be automatically chosen according to the dependencies stored along with the tests. Parallel running should be exploited whenever it is possible. Also, we think that supporting tools should not pose any constraint on the programming

language used to write test cases and they should provide a generic support for running network test cases written in any programming language.

Reporting. The tools should provide a clear report about the results of the run of the test cases.

Sharing. The tools should allow the user to search, download, and upgrade community shared network test cases and integrate them in the project test set in a similar way to what packaging systems allow Linux user to do with software package.

Configuration Management. The tools should allow to easily store and change configurations of network test cases.

Versioning. The tools should allow to track the history of tests and configurations and, possibly, revert to previous versions.

Coding. The tools should facilitate network administrators and operators in coding new network test cases as much as possible, providing a comprehensive set of API to support the implementation of common operations used in tests. As said before, no constraint should be posed on the programming language used to write test cases, hence, libraries should be provided in the most common languages. We believe that such libraries should be kept consistent as much as possible and be community shared.

Documentation. The tools should handle annotations that play the role of documentation providing a convenient way for browsing them.

Recoverability. The interruption of the execution (either by the user or due to an error) of a test case should be possible and must be safe for the network. While this is obvious for safe test cases, intrusive ones may provoke problems. The tools should provide supported procedures to recover from a network problem caused by an intrusive test case, restoring the initial status of the network.

We argue that a part of the network community can be interested in contributing to development and evolution of the supporting tools. For this reason, we think that the supporting tools should be implemented using widely known technologies, among which open and stable technologies are the most preferable. Moreover, the tools should incorporate a flexible mechanism to easily enrich API functionalities (for example, for offering new libraries to better support private coding of tests). From the point of view of the operator, the supporting tools should be very intuitive and simple to use. The idea is that operators can straightforwardly use the tools, through a clear user interface, both for managing test cases and running test suites. User interfaces provided should be both command based and graphic because the first are confortable for integration with other automatic tools, and the second are preferred for occasional use.

9 Related Work

Our methodology conforms to the ideas described in [15, 40], where it is argued that effective reactive testing is needed to deal with human errors, which are hardly addressable with proper design choices (like fault tolerant architectures commonly used for mitigating the risks of failures) and proactive approaches (e.g., better training or configuration doublechecking).

Some existent methodologies suggest to perform regression testing [46], which would be very useful for timely detecting human errors. However, except for major changes, regression testing is normally overlooked, since tests require to perform activities for preparation (e.g., manual arrangement of testing devices) and written documentation (e.g., test plans) which make testing activity unprofitable. For lowering costs of regression testing, some recent works propose to create cheaper testing environments, either exploiting virtualization techniques [7, 18, 42] or modifying the behavior of the network devices for managing testing and operational environments at the same time [5]. These approaches are complementary to TDND, since the possibility of creating virtual environments which exactly replicate operational networks eliminates the risks related to intrusive tests, making TDND more affordable and easier to be applied.

Some ideas about testing for human errors within the context of configuration management are also sketched in [2].

10 Conclusion and Future Work

In this paper, we have introduced the Test Driven Network Deployment methodology, which is mainly focused on supporting operators in timely detection of problems (in particular, human errors) to facilitate changes and evolution of enterprise networks. To reach this objective, the methodology stresses the role of testing, prescribing to execute large sets of in-depth tests after each change. For the methodology to be effective and affordable, the testing activity is required to be deeply automatized, grounding on easily manageable and repeatable tests and on convenient supporting tools. In this paper, we have outlined the main features of the tests that should be used in the methodology and we have provided some examples and requirements for supporting tools. Our future work will encompass the set up of a Web based community for sharing coarse-grained network test cases and the development of the supporting tools. We also plan to experimentally verify the effectiveness of TDND in real production environments.

References

[1] IEEE Standard for Software Unit Testing, Std 1008-1987. Technical report, IEEE Computer Society, 1986.

[2] Network Configuration Management. Cisco Systems white paper, 2007.

[3] What's Behind Network Downtime? Juniper networks white paper, May 2008.

[4] Ehab S. Al-Shaer and Hazem H. Hamed. Discovery of policy anomalies in distributed firewalls. In *INFOCOM*, 2004.

[5] Richard Alimi, Ye Wang, and Y. Richard Yang. Shadow Configuration as a Network Management Primitive. *SIGCOMM Comput. Commun. Rev.*, 38(4):111–122, 2008.

[6] Leopoldo Angrisani and Claudio Narduzzi. Testing Communication and Computer Networks: An Overview. *Instrumentation and Measurement Magazine, IEEE*, 11(5):12–24, October 2008.

[7] Andy C. Bavier, Nick Feamster, Mark Huang, Larry L. Peterson, and Jennifer Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *ACM SIGCOMM*, pages 3–14, 2006.

[8] Beck. *Test Driven Development: By Example.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[9] Kent Beck. Simple Smalltalk Testing: With Patterns. Smalltalk Report, October 1994.

[10] Kent Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.

[11] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, 1st edition, October 1999.

[12] Kent Beck and Erich Gamma. JUnit Test Infected: Programmers Love Writing Tests. Java Report, Volume 3, Number 7, July 1998.

[13] Rex Black. *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing.* John Wiley & Sons, Inc., New York, NY, USA, 2002.

[14] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[15] A. Brown and D. Patterson. Embracing failure: A case for recovery-oriented computing. In High Performance Transaction Processing Symposium, October, 2001, October 2001.

[16] B. Carpenter and S. Brim. Middleboxes: Taxonomy and issues, 2002.

[17] Cédric Teyssié and Zoubir Mammeri. UML-Based Approach for Network QoS Specification. In Pascal Lorenz and Petre Dini, editor, *ICN*, volume 3420 of *Lecture Notes in Computer Science*, pages 277–285. Springer, 2005.

[18] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, July 2003.

[19] John Clarkson, Caroline Simons, and Claudia Eckert. Predicting change propagation in complex design. In *ASME Journal of Mechanical Design*, volume 126, pages 765–797. ACM, 2004.

[20] Hakan Erdogmus. On the Effectiveness of the Test-First Approach to Programming. *IEEE Trans. Softw. Eng.*, 31(3):226–237, 2005. Member-Maurizio Morisio and Member-Marco Torchiano.

[21] Ernesto Exposito, Mathieu Gineste, Romain Peyrichou, Patrick Sénac, Michel Diaz, and Serge Fdida. XML QoS Specification Language for Enhancing Communication Services. In *ICCC '02: Proceedings of the 15th international conference on Computer communication*, pages 76–90, Washington, DC, USA, 2002. International Council for Computer Communication.

[22] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[23] Svend Frolund and Jari Koistinen. QML: A Language for Quality of Service Specification. Technical Report HPL-98-10, Hewlett-Packard Company, 1998.

[24] Ann Fruhling and Gert-Jan De Vreede. Field experiences with extreme programming: Developing an emergency response system. *J. Manage. Inf. Syst.*, 22(4):39–68, 2006.

[25] Jun Han. Supporting impact analysis and change propagation in software engineering environments. In *STEP '97: Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (STEP '97) (including CASE '97)*, page 172, Washington, DC, USA, 1997. IEEE Computer Society.

[26] Ahmed E. Hassan and Richard C. Holt. Predicting change propagation in software systems. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 284–293, Washington, DC, USA, 2004. IEEE Computer Society.

[27] Daniel Karlström. Introducing extreme programming - an experience report. In Proceedings of the Third International Conference on eXtreme Programming and Agile Processes in Software Engineering, pp. 24–29. Sardinia, Italy., 2002.

[28] Ramesh Kaza and Salman Asadullah. *Cisco IP Telephony: Planning, Design, Implementation, Operation, and Optimization (Networking Technology).* Cisco Press, 2005.

[29] Z. Kerravala. As the Value of Enterprise Networks Escalates, so does the Need for Configuration Management, January 2004.

[30] D. Richard Kuhn. Sources of failure in the public switched telephone network. *Computer*, 30(4):31–36, 1997.

[31] Martin Lippert, Henning Wolf, and Stefan Roock. *Extreme Programming in Action: Practical Experiences from Real World Projects.* John Wiley & Sons, Inc., New York, NY, USA, 2002.

[32] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-Testing: Unit Testing with Mock Objects. In *Extreme programming examined*, pages 287–301, Boston, MA, USA, 2001. Addison-Wesley Longman Publishing Co. Inc.

[33] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding bgp misconfiguration. *SIGCOMM Comput. Commun. Rev.*, 32(4):3–16, 2002.

[34] E. Michael Maximilien and Laurie Williams. Assessing Test-Driven Development at IBM. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 564–569, Washington, DC, USA, 2003. IEEE Computer Society.

[35] James D. McCabe. *Practical Computer Network Analysis and Design.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[36] Mikael Lofstrand and Jason Carolan. *Sun's Pattern-Based Design Framework: The Service Delivery Network*. Sun BluePrints OnLine, http://www.sun.com/blueprints/0905/819-4148.pdf, September 2005.

[37] Matthias M. Müller and Oliver Hagner. Experiment about test-first programming. *IEE Proceedings - Software*, 149(5):131–136, 2002.

[38] Peter G. Neumann. System and network trustworthiness in perspective. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 1–5, New York, NY, USA, 2006. ACM.

[39] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[40] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet services fail, and what can be done about it? In *Proc. of the 4th conf. on USENIX Symposium on Internet Technologies and Systems*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[41] Vasileios Pappas, Zhiguo Xu, Songwu Lu, Daniel Massey, Andreas Terzis, and Lixia Zhang. Impact of Configuration Errors on DNS Robustness. *SIGCOMM Comput. Commun. Rev.*, 34(4):319–330, 2004.

[42] Maurizio Pizzonia and Massimo Rimondini. Netkit: Easy emulation of complex networks on inexpensive hardware. In *Proc. International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom 2008)*, 2008.

[43] Priscilla Oppenheimer. *Top-Down Network Design (2nd Edition)*. Cisco Press, 2004.

[44] Václav Rajlich. Software change and evolution. volume 1725, pages 189–202, 1999.

[45] James Reason. *Human Error*. Cambridge University Press, 1990.

[46] Jr. Robert W. Buchanan. *The Art of Testing Network Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[47] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[48] Nancy Van Schooenderwoert. Embedded extreme programming: An experience report. Embedded Systems Conference Boston, 2004.

[49] Gary Stonenburner, Alice Goguen, and Alexis Feringa. Risk management guide for information technology systems. NIST Special Publication 800-30, January 2002.

[50] Lance Tatman. Incorporating Routing Analysis into IP Network Management. URL: http://www.agilent.com/labs/features/2003_wp_roca.pdf, May 2003.

[51] Diane Teare. *CCDA Self-Study: Designing for Cisco Internetwork Solutions (DESGN) 640-863, Second Edition*. Cisco Press, 2007.

[52] Christian Webel and Reinhard Gotzhein. Formalization of network quality-of-service requirements. In John Derrick and Jüri Vain, editors, *FORTE*, volume 4574 of *Lecture Notes in Computer Science*, pages 309–324. Springer, 2007.

[53] W. Willinger and J. Doyle. Robustness and the internet: Design and evolution, 2002.