Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Testing Peer-to-Peer Systems

the date of receipt and acceptance should be inserted later

Abstract Peer-to-peer (P2P) offers good solutions for many applications such as large data sharing and collaboration in social networks. Thus, it appears as a powerful paradigm to develop scalable distributed applications, as reflected by the increasing number of emerging projects based on this technology. However, building trustworthy P2P applications is difficult because they must be deployed on a large number of autonomous nodes, which may refuse to answer to some requests and even leave the system unexpectedly. This volatility of nodes is a common behavior in P2P systems and may be interpreted as a fault during tests (i.e., failed node). In this work, we present a framework and a methodology for testing P2P applications. The framework is based on the individual control of nodes, allowing test cases to precisely control the volatility of nodes during their execution. We validated this framework through implementation and experimentation on an open-source P2P system. The experimentation tests the behavior of the system on different conditions of volatility and shows how the tests were able to detect complex implementation problems.

Keywords Software Testing · Peer-to-peer systems · Distributed hash tables (DHT) · Testing methodology · Experimental procedure

1 Introduction

P2P appears as a powerful paradigm to develop scalable distributed systems, as reflected by the increasing number of projects based on this technology (Androutsellis-Theotokis and Spinellis [2004]). Among the many aspects of P2P development, producing systems that work correctly is an obvious target. This is even more critical when P2P systems are to be widely used. Thus, as for any system, a P2P system should be tested with respect to its requirements. As for any distributed system, the complexity of message exchanges must be a part of the testing objectives. Testing of distributed systems typically consists of a centralized test architecture composed of a test controller, or coordinator, which synchronizes and coordinates communication (message calls, deadlock detection) and creates the overall verdict from the local verdicts. Local to each node, test sequences or test automata can be executed, which run these

Address(es) of author(s) should be given

partial tests on demand and send their local verdicts to the coordinator. One local tester per node or group of nodes is generated from the testing objectives. Distributed systems are commonly tested using conformance testing (Schieferdecker et al. [1998]). The purpose of conformance testing is to determine to what extent the implementation of a system conforms to its specification. The tester specifies the system using Finite State Machines (Chen and Ural [1995]; Hierons [2001]; Chen et al. [2006]) or Labeled Transition Systems (Jard [2001]; Pickin et al. [2002]; Jard and Jéron [2005]) and uses this specification to generate a test suite that is able to verify (totally or partially) whether each specified transition is correctly implemented. The tester then observes the events sent among the different system nodes and verifies that the sequence of events corresponds to the specification.

In a P2P system, a peer plays the role of an active node with the ability to join or leave the network at any time, either normally (e.g., disconnection) or abnormally (e.g., failure). This ability, which we call volatility, is a major difference with distributed systems. Furthermore, volatility yields the possibility of dynamically modifying the network size and topology, which makes P2P testing quite different. Thus, the functional behavior of a P2P system (and functional flaws) strongly depends on the number of peers, which impacts the scalability of the system, and their volatility.

As an illustration, Distributed Hash Table (DHT) (Rowstron and Druschel [2001]; Ratnasamy et al. [2001]; Stoica et al. [2001]) is a basic P2P system, where each peer is responsible for the storage of values corresponding to a range of keys. A DHT has a simple local interface that only provides three operations: value insertion, value retrieval and key look-up. The remote interface is more complex, providing operations for data transfer and maintenance of the routing tables, i.e., the correspondence table between keys and peers, used to determine which peer is responsible for a given key. Considering the simplicity of the interface, testing a DHT in a stable system is quite simple, but does not provide any confidence in the correctness of implementation for the specific P2P mechanisms. When peers leave and join the system, the test must check that both the routing table is correctly updated and that requests are correctly routed.

In this paper, we present a framework for testing peer-to-peer systems, including testers and coordinator, with the ability to create peers and make them join and leave the system. With this framework, the test objectives can combine the functional testing of the system with the volatility variations (and also scalability). The correctness of the system can thus be checked based on these three dimensions, i.e., functions, number of peers and volatility. We present an incremental methodology to deal with these dimensions, which aims at covering functions first on a small system and then incrementally addressing the scalability and volatility aspects. Empirical results obtained by running several test cases illustrate the fact that satisfying a simple test criterion such as code coverage is a hard task. Open issues, such as the generation of efficient test objectives are also identified.

The rest of the paper is organized as follows. The next section introduces the basic concepts and proposes a testing methodology. Section 3 presents our framework for P2P testing. Section 4 presents how to write a test case using our framework. Section 5 describes our validation through implementation and experimentation on an open-source P2P system. Section 6 discusses related work. Section 7 concludes.

2 Basic concepts

Software testing verifies a system dynamically, observing its behavior during the execution of a suite of *test cases*. The objective of a test case is to verify if a feature is correctly working according to certain quality criteria: robustness, correctness, completeness, performance, security, etc. Typically, a test case is composed of a name, an intent, a sequence of input data including a preamble, and the expected output. In this section, we introduce the basic concepts and requirements for a P2P testing framework. Moreover, we present a testing methodology.

2.1 Requirements for a testing framework

As stated in the introduction, P2P testing tackles the classical issue of testing a distributed system, but with a specific dimension which we call *volatility*, which has to be an explicit parameter of the test objectives. Two possible solutions may be used to obtain a test sequence which includes volatility. It can either be simulated with a simulation profile or be explicitly and deterministically decided in the test sequence. The first solution is the easiest to implement, by assigning a given probability for each peer to leave or join the system at each step of the test sequence execution. The problem with this approach is that it makes the interpretation of the results difficult, since we cannot guess why the test sequence failed. Moreover, it creates a bias with the possible late responses of some peers during the execution of the test sequence. For instance, a peer may left the system or experienced a fault. As a result, it cannot be used to combine a semantically rich behavioral test with the volatility parameter. In this paper, we recommend to fully control volatility in the definition of the test sequence. Thus, a peer, from a testing point of view, can have to leave or join the system at a given time in a test sequence. This action is specified in the test sequence in a deterministic way.

Since we must be able to deal with large numbers of peers, the second dimension of P2P system testing is scalability. Because it is accomplishing a treatment, the scalability and volatility dimensions must be tested with behavioral and functional correctness. In summary, a P2P testing framework should provide the possibility to control:

- functionality captured by the test sequence TS which enables a given behavior to be exercised,
- scalability captured by the number p of peers in the system,
- volatility captured by the number v of peers which leave or join the system after its initialization during the test sequence.

To test a P2P system, we need a framework which allows a test to be both specified and executed in the entire system. This means that at each peer, a local tester must be deployed to control the local-to-a-peer execution. This also means that a centralized node must guide and control the overall executions and verdicts of the local testers. We call such a test as global test. Thus, the framework should be consistent with the UML deployment diagram of Figure 1, which inserts *tester* components as well as a *coordinator*.

The coordinator controls several testers and each tester runs on a different logical node (the same as the peer it controls). The role of the tester is to execute test case actions and to control the volatility of a single peer. The role of the coordinator is to dispatch the actions of a test case (A^τ) through the testers (T^τ) and to maintain a list

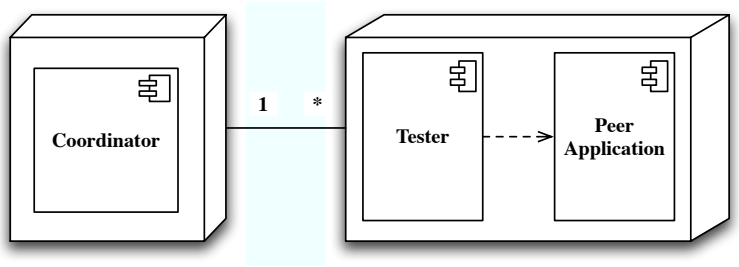


Fig. 1 Deployment Diagram

of unavailable peers. In practice, each tester receives the description of the overall test sequence and is thus able to know when to apply a local execution sequence.

2.2 Testing methodology

When testing scalability of a distributed system, the functional aspects are typically not taken into account. The same basic test scenario is simply repeated on a large number of nodes (Duarte et al. [2006]). The same approach may be used for volatility, but would also lead to test volatility separately from the functional aspect. For a P2P system, we claim that the functional flaws are strongly related to the scalability and volatility issues.

This because functionalities are specially designed to work with a variable number of nodes (from one up to more than one million) and with the arrival and the departure of nodes. Functionalities do not perform the same in a small system, where each node knows every other nodes, as they perform on large systems, where each node only have a partial view of the whole system. In the last case, the accomplishment of a functionality often leads to more complex communications, such as message routing or node discovery.

When nodes leave or join the system, different functions are performed, other than the update of the routing tables. For instance, in many distributed hash tables, when a node joins the system, it becomes responsible for a range of keys. Thus, before starting to respond to queries, it must receive from other nodes all data associated to these keys. And when the node leaves the system, the inverse transfer of data must be done.

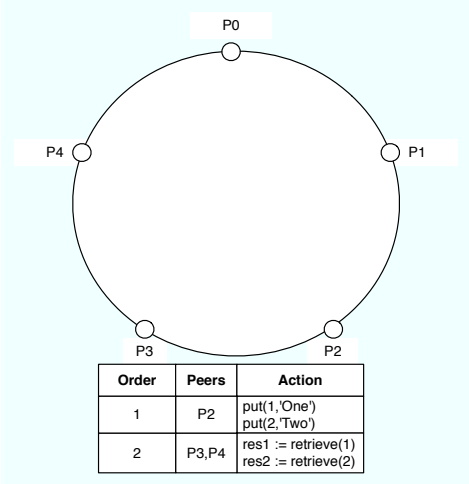
Therefore, it is crucial to combine the scalability and volatility aspects with meaningful test sequences. To take into account the three dimensional aspects of P2P systems, we present a methodology that combines the functional testing of a system with the variations of the other two aspects. Indeed, we incrementally scale up the SUT either simulating or not volatility. This simulation can be executed with different workloads, such as: shrinking the system, expanding it or both at the same time. These different workloads may exercise different behaviors of the SUT and possibly reveal different flaws.

Our incremental methodology is composed by the following steps:

1. small scale application testing without volatility;
2. small scale application testing with volatility;
3. large scale application testing without volatility;
4. large scale application testing with volatility.

Step 1 consists of conformance testing, with a minimum configuration. The goal is to provide a test sequence set efficient enough to reach a predefined test criteria. These test sequences must be parameterized by the number of peers $TS(P)$, so that they can be extended for large scale testing. Test sequences can also be combined to build a complex test scenario using a test language such as Tela (Pickin et al. [2001]).

In our motivating example, we start a stable system with all the peers set as illustrated in Figure 2. The peer p_2 will insert some data into a DHT, then the peers p_3 and p_4 will retrieve them. This first step aims to verify pure functional problems without interference with the size of the system and/or volatility. In the case of a stable and small scale DHT, all the peers probably know each other representing minimal or even nonexistent routing table updates. Thus, messages may be exchange directly between peers.

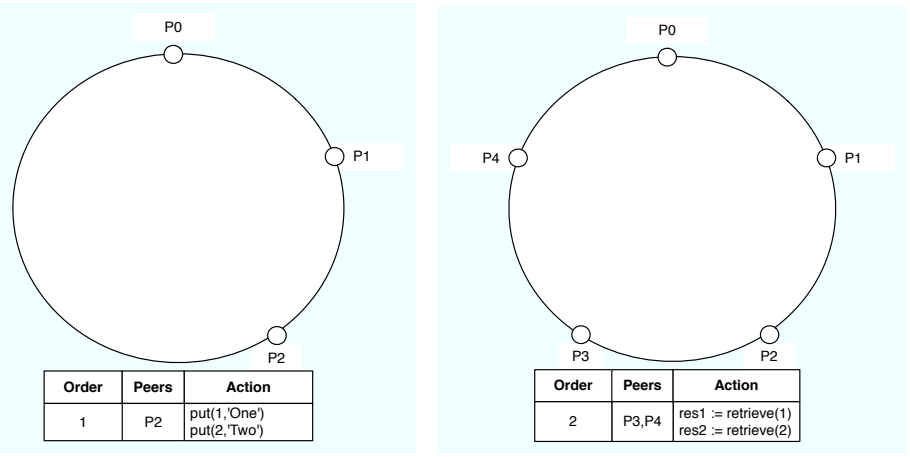


Order	Peers	Action
1	P2	put(1,'One') put(2,'Two')
2	P3,P4	res1 := retrieve(1) res2 := retrieve(2)

Fig. 2 Small scale application testing without volatility (Step 1)

Step 2 consists of reusing the initial test sequences and adding the volatility dimension. The result is a set of test sequences including explicit volatility (TSV). Figure 3(a) illustrates a DHT before volatility when data is inserted by peer p_2. Then, the peers p_3 and p_4 join the system and retrieve data as illustrated in Figure 3(b). This second step aims to verify functional problems related to volatility at a small scale considering that pure functional problems were isolated at Step 1. Indeed, testing inserts and retrieves upon volatility exercises both data forwarding and routing table update. Furthermore, a small scale system guarantees low forwarding since data tend to be sent to peers within the routing table.

Step 3 reuses the initial test sequences of Step 1 combining them to deal with a large number of peers. We thus obtain a global test scenario GTS. A test scenario composes test sequences. This third step aims to verify functional problems related to scalability. To do so, we test the SUT without volatility in a large scale. As described in Step 1, a stable system represents minimal or even nonexistent routing table updates. Whenever we scale up the SUT, peers are obligated to perform some tasks like routing



(a) DHT before volatility (b) DHT after volatility

Fig. 3 Small scale application testing with volatility (Step 2)

messages and forwarding data to unknown peers. Indeed, these tasks could be only tested in large scale systems since peers are unlikely to know all the others.

Figure 4 illustrates a large scale and stable DHT. In our motivating example, peer p_2 inserts some data into the DHT respectively at p_1 and p_2. Whenever the peers p_3 and p_4 try to retrieve data, they probably do not know p_1 and p_2, messages are routed until reach such data. Therefore, aspects related to scalability, such as message routing, can be verified from this third step.

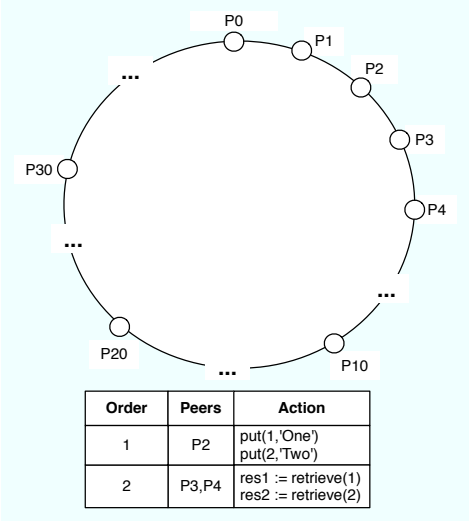
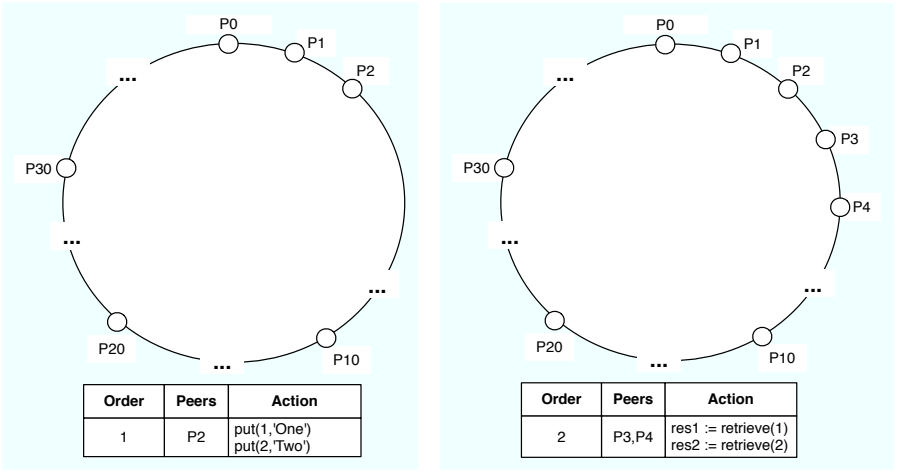


Fig. 4 Large scale application testing without volatility (Step 3)

Step 4 reapplies the test scenarios of Step 3 with the test sequences of Step 2, and a global test scenario with volatility ($GTSV$) is built and executed. Figure 5 illustrates a large scale DHT upon volatility. In fact, this step aims to verify the problems related to all three dimensions. Therefore, after the insertion of data illustrated in Figure 5(a), peers come and go depending on the type of the volatility. For simplicity, Figure 5(b) illustrates the join of new peers p_3 and p_4. In our example, the successors of both p_3 and p_4 have to update their routing table and route messages. Eventually, the test case can be improved to store something at p_3 or p_4 in order to exercise data forwarding as well.



(a) DHT before volatility (b) DHT after volatility

Fig. 5 Large scale application testing with volatility (Step 4)

The advantage of this process is to focus on the generation of relevant test sequences, from a functional point of view, and then reuse these basic test sequences by including volatility and scalability. The test sequences of Step 1 satisfy test criteria (code coverage, interface coverage). When reused at large scale, the test coverage is thus ensured by the way all peers are systematically exercised with these basic test sequences.

In terms of diagnosis, this methodology allows to determine the nature of the detected erroneous behavior. Indeed, the problem can be linked to a purely functional cause (Step 1), a volatility issue (Step 2), a scalability issue (Step 3) or a combination of these three aspects (Step 4). The most complex errors are the last ones since their analysis is related to a combination of the three aspects. Steps 2 and 4 could also be preceded by two other steps (shrinkage and expansion), to help the diagnosis of errors due to either the unavailability of resources or arrival of new ones. Yet, several rates of volatility can be explored to verify how they affect the functionality aspect of the SUT (e.g., 10% joining, 20% leaving).

2.3 Definitions

Let us denote by P the set of peers representing the P2P system, which is the system under test (SUT). We denote by T, where $|T| = |P|$ the set of testers that controls the SUT, by DTS the suite of tests that verifies P, and by A the set of actions executed by DTS on P.

Considering that all peers have exactly the same interface, testing the interface of a single peer is not sufficient to test the whole system. Actually, multiple peers are needed to simulate the P2P behavior, with each peer potentially executing a different part of the test case. This is why we introduce the notion of distributed test cases, i.e., test cases that apply to the whole system and whose actions may be executed by different peers.

Definition 1 (Distributed test case) A distributed test case noted τ is a tuple $\tau = (A^\tau, T^\tau, L^\tau, S^\tau, V^\tau, \varphi^\tau)$ where $A^\tau \subseteq A$ is an ordered set of actions $\{a_0^\tau, \ldots, a_n^\tau\}$, $T^\tau \subseteq T$ a set of testers, L^τ is a set of local verdicts, S^τ is a schedule, V^τ is a set of variables and φ^τ is the level of acceptable *inconclusive* verdicts.

The Schedule is a map between actions and sets of testers, where each action corresponds to the set of testers that execute it.

Definition 2 (Schedule) A schedule is a map $S : A \mapsto \Pi$, where Π is a collection of tester sets $\Pi = \{T_0, \ldots, T_n\}$, and $\forall T_i \in \Pi : T_i \subseteq T$.

In P2P systems, the autonomy and the heterogeneity of peers interfere directly in the execution of service requests. While close peers may answer quickly, distant or overloaded peers may need a considerable delay to answer. Consequently, clients do not expect to receive a complete result, but the available results that can be retrieved during a given time. Thus, test case actions (Definition 3) must not wait indefinitely for results, but specify a maximum delay (*timeout*) for an execution.

Definition 3 (Action) A test case action is a tuple $a_i^\tau = (\Psi, \kappa, T')$ where Ψ is a set of instructions, κ is the interval of time in which Ψ should be executed and $T' \subseteq T$ is a set of testers that executes the action.

The instructions are typically calls to the peer application interface as well as any statement in the test case programming language.

A test sequence is a sequence of messages (e.g., service requests) exchanged among peers along testing.

Definition 4 (Test sequence) A test sequence is a sequence of test actions $TS(A)$, where each action a_n^A has a hierarchical level $h_{a_n}^A$. Actions with lower levels are executed before actions with higher levels.

Definition 5 (Local verdict) A local verdict is given by comparing the expected result, noted E, with the result itself, noted R. E and R may be a single value or a set of values from any type. However, these values must be comparable. The local verdict v of τ within κ is defined as follows:

$$l_\kappa^\tau = \begin{cases} pass & \text{if } R = E \\ fail & \text{if } R \neq E \\ inconclusive & \text{if } R = \emptyset \end{cases}$$

2.4 Test case example

Let us illustrate these definitions with a simple distributed test case (see Example 1). The aim of this test case is to detect errors on a DHT implementation. More precisely, it verifies whether new peers are able to retrieve data inserted before their arrival.

Example 1 (Simple test case)

Action	Testers	Action
(a_1)	0,1,2	join()
(a_2)	2	Insert the string "One" at key 1; Insert the string "Two" at key 2;
(a_3)	3,4	join();
(a_4)	3,4	Retrieve data at key 1; Retrieve data at key 2;
(a_5)	*	leave();
(v_0)	0	Calculate a local verdict;
(v_1)	1	Calculate a local verdict;
(v_2)	2	Calculate a local verdict;

This test case involves five testers $T^\tau = \{t_0 \dots t_4\}$ that control five peers $P = \{p_0 \dots p_4\}$ and five actions $A^\tau = \{a_1^\tau, ..., a_5^\tau\}$. If the data retrieved in a_4 is the same as the one inserted in a_2, then the verdict is $pass$. If the data is not the same, the verdict is $fail$. If t_3 or t_4 are not able to retrieve any data, then the verdict is *inconclusive* (e.g., action timeout). For each tester a local verdict is calculated and send to a test coordinator (discussed later on Section 3.2).

3 P2P testing framework

The framework was implemented in Java (version 1.5) and makes extensive use of two Java features: dynamic reflection and annotations. As we will see in the following, these features are used to select and execute the actions that compose a test case. Our objective when developing the framework was to make the implementation of test cases as simple as possible. The framework is not driven by a specific type of tests (e.g., conformance, functional, etc.) and can be used as a basis for developing different test cases. The framework has two main components: the tester and the coordinator. The tester is composed of the test suites that are deployed on several logical nodes. The coordinator is deployed in only one node and is used to synchronize the execution of test cases. It acts as a *broker* (Buschmann et al. [New York, NY, 1996]) for the deployed testers. The framework is developed as a distributed application using Java-RMI. Testers are not expected to leave and join the system, contrarily to the peers.

3.1 Testers

In general, a tester is the control/observation component responsible to manage test suites (i.e., test cases and test sequences). A test suite is implemented as a class,

which is the main class of the testing application. A test suite contains several test cases, which are implemented as a set of actions. Test case actions are implemented as *annotated* methods, i.e., methods adorned by a particular meta-tag, or *annotation*, that informs that the method is a test case action, among other information. Annotations can be attached to methods and to other elements (e.g., packages, types, etc.), giving additional information concerning an element: the class is deprecated, a method is redefined, etc. Furthermore, new annotations can be specified by developers. Method annotations are used to describe the behavior of test case actions: where it should execute, when, in which tester, whether or not the duration should be measured. The annotations are similar to those used by JUnit[1], although their semantics are not exactly the same.

The choice of using annotations for synchronization and conditional execution was motivated to be consistent with JUnit's structure to accelerate industry acceptance. Additionally, the annotations are helpful for three reasons. First, to separate the execution control from the testing code. The execution control is a duty of the framework and should be done automatically upon certain rules or parameters. These parameters are provided by the annotations and filled by a test engineer while writing the testing code. Second, to simplify the deployment of the test cases. One approach would deploy the precise set of actions to each tester. However, such deployment could be expensive in a very large system since different testers manage different sets of actions. In our approach, all testers receive the same test case, however, they only manage the actions annotated to them. Third, to implement a testing system that supports effective and repeatable automated testing also called as test harness (Binder [1999]). The test harness requires four steps:

1. **Set up:** sets the SUT to be tested. This means the runtime environment is set. This step is also called as preamble scenario. For instance, loading databases, files or memory; establishing the peer set within a DHT.
2. **Execute:** exercises the SUT by inputs and outputs that are stored to be compared afterward. For instance, inserting data into the DHT, then retrieving such data.
3. **Evaluate:** evaluates a test case execution. This step is the oracle implementation. Thus, the test case must compare the actual results with the expected ones and the results can be both displayed and/or logged.
4. **Clean up:** releases all the resources used within testing. The idea is to let the SUT in the same state as before the test case begins. Its purpose is to ensure that there will be no interference among different test cases. This step is also called as postamble scenario. For instance, stopping databases; closing network connections; making all the peers leave a DHT.

In our framework, all the testers receive the same test case to simplify the deployment of a distributed test case. However, the testers only execute the actions assigned to them in agreement with the annotations. The available annotations are listed below:

Test: specifies that the method is actually a test case action. This annotation has four attributes that are used to control its execution: the test case name, the place where it should be executed, its order inside the test case and the execution timeout.

Before: specifies that the method is executed before each test case. The purpose of this method is to set up a common context for all test cases. The method plays the role of a preamble scenario.

[1] http://www.junit.org

After: specifies that the method is executed after each test case. The purpose of this method is to clean up the resources used along testing. The method plays the role of a postamble.

Each action is a point of synchronization: at a given moment, only methods with the same signature can be executed on different testers. Actions are not always executed on all testers, since annotations are also used to restrain the testers where an action can be executed. Thus, testers may share the same testing code but do not have the same behavior. The purpose is to separate the testing code from other aspects, such as synchronization or conditional execution. The tester provides two interfaces, for action execution and volatility control:

1. **execute**(a_i): executes a given action.
2. **leave(), fail(), join()**: makes a peer leave, abnormally quit or join the system.

When a tester executes a test case, it proceeds the following steps:

1. It asks the coordinator for identification, used to filter the actions that it should execute. In this step, the coordinator starts the registration of testers.
2. It uses java reflection to sort out the actions, read their annotations and create a set of method descriptions (i.e., the signature of methods it should manage).
3. It passes to the coordinator the set of method descriptions that it should manage and their priorities. In this step, the coordinator finishes the registration of testers.
4. It waits for the coordinator to invoke one of its methods. After the execution, it informs the coordinator that the method was correctly executed.

Figure 6 illustrates the fourth step of the test case execution, more precisely the execution of action a_2 from Example 1. First, the coordinator acknowledges that action a_2 should start. Second, it seeks out the tester ids for a_2 within the schedule. In our example, a_2 is set to be executed by t_2. Then, t_2 receives a message from the coordinator to execute a_2 in peer p_2. Finally, the action is executed and its end is sent back to the coordinator to restart this step to the next action. The test case execution is detailed in Section 3.3.

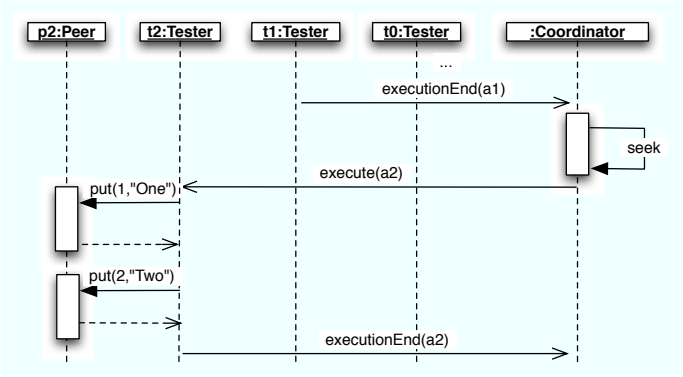


Fig. 6 Test case execution

3.2 The Coordinator

The coordinator is a central component that controls when each test action should be executed. In fact, it coordinates the execution of actions using the schedule. To do so, it builds the schedule along the registration of testers (i.e., testers requiring identification) as illustrated in Figure 7. Two testers register their actions while the coordinator updates the schedule with their ids and corresponding actions. The construction of the schedule is detailed later in Section 3.3.

Once the registration is over, the coordinator is able to start the execution of the actions. When the test starts, the coordinator receives a set of method descriptions from each tester and associates each action to a hierarchical level. Such hierarchical level is used to sort the actions and guide their execution. Then, it iterates over a counter representing the hierarchical level that can be executed, allowing the execution to be synchronized.

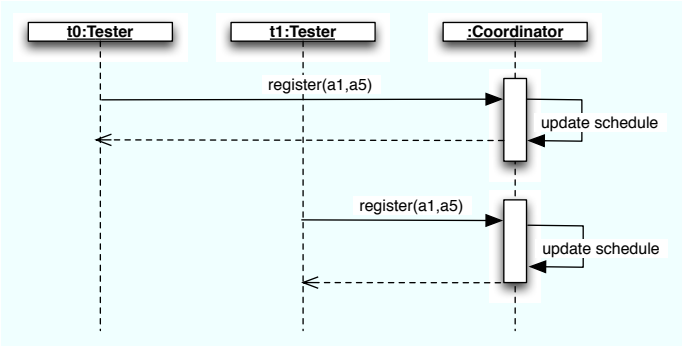


Fig. 7 Schedule construction

The execution of actions follows the idea of two phase commit (2PC). In the first phase, the coordinator informs all the the concerned peers that an action a_n can be executed and produces a lock. Once all peers announce the end of their execution, the lock is released and the execution of the next action begins. If an action's timeout is reached, the test case is *inconclusive*.

The coordinator provides three different interfaces, for action execution, volatility and test case variables (described in Section 3.6):

- **register**(t_i, A^t)**, ok**(a_n)**, fail**(a_n)**, error**(a_n): performs the actions registration (performed before all tests) and response for actions execution, called by testers once the execution of an action is finished.
- **set(key,value), get(key)**: provides the accessors for test case variables.
- **leave**(P)**, fail**(P)**, join**(P): makes a set of peers leave, abnormally quit or join the system.

3.3 Test case execution

The synchronization algorithm has three steps (see Algorithm 1): registration, action execution and verdict construction. Before the execution of a τ, each $t \in T$ registers its actions with the *coordinator*. For instance, in the motivating example (see Example 1), tester t_2 may register the actions $A' = \{a_1, a_2, a_5\}$.

Algorithm 1: Test suite execution

```
   Input: T, a set of testers; DTS, a distributed test suite
   Output: Verdict
 1 foreach t ∈ T do
 2 |  register(t);
 3 end
 4 foreach τ ∈ DTS do
 5 |  foreach a ∈ A^τ do
 6 |  |  foreach t ∈ S^τ(a) do
 7 |  |  |  send execute(a) to t;
 8 |  |  end
 9 |  |  wait for an answer from all t ∈ (S^τ(a) − T_u) ;
10 |  end
11 |  foreach t ∈ T^τ do
12 |  |  L^τ ← L^τ + l_t^τ ;
13 |  end
14 |  return oracle(L^τ, φ) ;
15 end
```

The registration algorithm works as follows (see Algorithm 2). Initially, the *coordinator* identifies each *tester* with an integer identifier, for instance the first tester receives the identifier 0. The identifier is increased by 1 every time a new tester asks for it. This simple method simplifies the action definition that is made at the user level. The identifier is also used by a node to know whether it is allowed to execute a given action.

Algorithm 2: Registration

```
   Input: p, a node
   Output: id
1 foreach a ∈ A^p do
2 |  S^τ(a) ← S^τ(a) + p ;
3 end
4 id++ ;
5 return id ;
```

Once the registration is finished, the *coordinator* builds the schedule S, mapping the actions with their related subset of testers. In our example, action a^3 is mapped to $\{t_3, t_4\}$. Once S is built, the coordinator traverses all test cases $\tau \in DTS$ and then the actions of each τ. For each action a_i^τ, it uses $S^\tau(a_i^\tau)$ to find the set of testers that are related to it and sends the asynchronous message $execute(a_i) \forall t \in S^\tau(a_i^\tau)$. Then, the coordinator waits for the available testers to inform the end of their execution. The set of available testers corresponds to $S^\tau(a_i^\tau) - T_u$, where T_u is the set of unavailable

testers. In our example, once a_1 is finished, testers $\{t_0, t_1, t_2\}$ inform the *coordinator* of the end of the execution.

Algorithm 3: Action execution

```
Input: a, an action to be executed
invoke(a) ;
if κ is reached then
    send error to Coordinator ;
else
    send ok to Coordinator ;
end
```

Thus, the *coordinator* knows that a_1 is completed and the next action can start. When a tester $t \in T^\tau$ receives the message $execute(a_n^\tau)$, it executes the suitable action. Each tester t_i receives the asynchronous message $execute(a)$ and then performs action a as described in algorithm 3. If the execution succeeds, then a message *ok* is sent to the coordinator. Otherwise, if the action timeout is reached, then the message *error* is sent.

Once the execution of τ finishes, the coordinator asks all testers for a local verdict. In the example, if t_3 gets the correct strings "One" and "Two" at a_4, then its local verdict is $pass$. Otherwise, it is $fail$. After receiving all local verdicts, the coordinator is able to assign a verdict L^τ. If any local verdict is $fail$, then L^τ is also $fail$, otherwise the coordinator continues grouping each l_t^τ into L^τ. When L^τ is completed, it is analyzed to decide between verdicts $pass$ and $inconclusive$ as described in Algorithm 4. This algorithm has two inputs, a set of local verdicts (L) and an index of relaxation (φ), representing the level of acceptable *inconclusive* verdicts (detailed in Section 3.5). If the ratio between the number of $pass$ and the number of local verdicts is greater than φ, then the verdict is $pass$. Otherwise, the verdict is $inconclusive$.

Algorithm 4: Oracle

```
Input: L, a set of local verdicts; φ an index of relaxation
if ∃l ∈ L, l = fail then
    return fail
else if |{l ∈ L : l = pass}|/|L|) ≥ φ then
    return pass
else
    return inconclusive
end
```

3.4 Dealing with node volatility

The volatility of nodes can make testing difficult during the execution of a test case. If the coordinator is not informed that a node has left the system, then it is unable to follow the test sequence and is unable to proceed. We must then be able to control node volatility to forecast the next action that must be performed. Our algorithm treats the

volatility of nodes as common actions, where the tester informs the coordinator that a node has joined/left the system.

Since the coordinator is informed by the testers of node departures or fails, it is able to update its schedule and does not wait for confirmation from these nodes. Therefore, the next action is set for execution and the synchronization sequence continues.

3.5 Setting a global verdict

To set a global verdict, the algorithm should also take into account the autonomy of nodes, since it directly influences the result completeness of queries. Result completeness guarantees that a result set is complete with respect to the set of objects in its source. For instance, in a distributed database system, query results are complete since all nodes are expected to answer.

In a P2P system, when a node queries the system a partial result set may satisfy the request. As some nodes may not answer (e.g., due to timeout constraints), there is no guarantee of completeness. During the verification of a P2P system, the lack of result completeness may engender *inconclusive* verdicts, since the oracle can not state if the test case succeeded or not. This lack of completeness should not be interpreted as an error, it belongs to the normal behavior of P2P systems.

Thus, we introduce an index for completeness relaxation, which was showed in algorithm 4 as φ. This index is used to take into consideration *inconclusive* verdicts engendered by lack of response to some node request. It represents the percentage of desirable *pass* verdicts in V^{τ}. Such index is chosen by the testing designer, since it represents the human knowledge about the SUT completeness.

It is also possible to avoid *inconclusive* verdicts by other means, such as: relaxing time constraints, making tighter constraints or even adjusting the sequence of actions. However, one may consider the pros and cons for each one. Relaxing time constraints would lead to less *inconclusive* verdicts depending on the time frame set. In this case, the testing environment may influence this constraint since some conditions might change while shifting to another environment. For instance, the time of a message round trip in a grid machine is much lower than in a local network. Then, the same test case may $fail$ or *pass* depending on the environment. This tends to be the same whether making a tighter time constraint.

Making tighter constraints would also lead to less *inconclusive* verdicts, but in some case would lead to false-negative/false-positive verdicts. For instance, setting a smaller number of peers would result in smaller routing of messages or even none routing at all, then bugs related to the size of the system would not be detected.

Finally, adjusting the sequence of actions would also help avoiding *inconclusive* verdicts. In this case, some adjusts have to be constrained to ensure the correctness of the test cases. For instance, the P2P system must be started before testing any other service, otherwise, the test case tends to be useless.

3.6 Test case variables

We call *Test Case Variables* a structure that keeps values used along testing and cannot be predicted when a test case is written. This means that values generated on-the-fly can be stored and used at runtime. To access this structure, kept by the central coordinator, the framework provides an interface to insert and retrieve variables as described at the testers and coordinator section.

Several testing scenarios can take advantage of this structure. For instance, testers must analyze the routing table of their peers to verify if it was correctly updated. More precisely, testers must compare the peer IDs from a routing table with the ID of peers that leave or join the system. This comparison is not trivial, because each tester only knows the ID of its peer, which is dynamically assigned. For instance, a given tester may be aware that the peers controlled by testers t_0 and t_1 left the system, but does not know that they correspond to peers $20BD8AB6$ and $1780BB16$. To simplify the analysis of routing tables, we use test case variables to map tester IDs to peer IDs, as shown in Table 1.

Tester ID	**Peer ID**
0	20 BD 8A B6
1	17 80 BB 16
2	A0 36 02 F7
3	40 E4 DE A2
...	...
15	FA 09 EE 90
16	21 3A C2 58

Table 1 ID mapping

The variables are stored only at runtime and must be purged by the end of each test case execution. This avoids interference with further executions and happens within the postamble scenario.

4 Writing test cases

In this section, we present how to write a test case using our framework. First, we describe the annotations used by the testers to manage the test execution (i.e., synchronization and deployment). Second, we describe the assertions provided by the framework. Third, we describe a simple test case wrote over the motivating example (see Example 1).

4.1 Annotations

The framework provides three annotations. The first annotation specifies that a method is a test case action. As mentioned, a test case action is a point of synchronization used to manage the test sequence. This annotation, called @Test, provides the following parameters:

– "testers" informs the id of a tester responsible to manage the action on a peer. Note that the testers' id are dynamically assigned during the test case registration, and

are not related to the peers' id, which are proper to the SUT. If set to "$*$", then all the testers will manage the action. It is also possible to inform sets of tester ids to manage the action. For instance, "$0-9$" will make these 10 testers to run the action, or "$0-9, 20-29$" will make two sets of testers to run all together. It is also possible to inform a set of testers separated by commas, like this "$0-11, 34, 47$".
- "name" informs the name of the test case. Different methods may be part of the same test case.
- "step" informs that a test case has different steps during the execution.
- "measure" measures the execution of the action in milliseconds.
- "timeout" indicates a milliseconds time frame to execute the action. If this time expires, then it is assigned a local *inconclusive* verdict.

The code below shows the annotation usage. This action, that belongs to test case "tc1", will be managed by testers 0 to 20. The other parameters indicate that the execution will be measured and it has 1000 milliseconds to finish.

```
@Test(testers="0-20", name="tc1", step=1, measure=true, timeout=1000)
public void action(){
    ...
}
```

The second and third annotations specify the test case set up and clean up respectively. The objective is to comply with the test harness described in Section 3.1. These annotations, called @Before and @After, provide some of the same parameters used by @Test, such as: "testers" and "timeout".

The code below shows the @Before and @After usage. These methods will be managed by all the testers since the "testers" parameter value is set to "*". The execution of each method has a time frame of 1000 milliseconds.

```
@Before(testers="*", timeout=1000)
public void setupAction(){
    ...
}
...
@After(testers="*",timeout=1000)
public void cleanAction(){
    ...
}
```

4.2 Assertions

Assertions provide a powerful and straightforward test oracle approach that can be provided either by a programming language (e.g., Java, Eiffel, C++) or by a testing framework (e.g., Junit, JTiger, TestNG, GNU Nana and APP (Rosenblum [1995])). The difference is that a testing framework can be used to develop a test application, while the assertions of programming languages are built inside the SUT's source code and used to check required constraints.

Our framework inherits the default assertion methods from the JUnit framework's TestCase class. Some of the most important assertions are:

- **assertTrue(boolean condition)** asserts if a condition is true.
- **assertFalse(boolean condition)** asserts if a condition is false.

- **fail(String message)** fails a test with the given message.
- **assertEquals(String message, Object expected, Object actual)** asserts that two objects are equal.
- **assertArrayEquals(String message, Object[] expecteds,Object[] actuals)** asserts that two object arrays are equal.

Our framework also offers two additional test assertions:

- **inconclusive(String message)** indicates an *inconclusive* verdict. This assertion can be used within a test case to assign a verdict in case some situation hinders the flow of the test case, but can not assign a fail (e.g., a peer could not join the system). It is also used internally by the framework in case of a test case execution's timeout.
- **assertCollectionEquals(String message,Collection expecteds, Collection actuals)** asserts that two collections are equal.

In the assertions presented above except for **fail(String message)** and **inconclusive(String message)**, if the condition is not met, the assertion throws an error. In this case, the oracle assigns the *fail* verdict. A code example is provided below.

The code below verifies if a peer can join the system during the start up. This follows the standard JUnit's structure which provides a straightforward manner to assure the test harness. If such peer cannot join the system, then the peer cannot proceed the test case execution. In this case, it is not possible to indicate that the system is buggy. In fact, it only indicates that an unmanageable situation occurs, then an *inconclusive* verdict is assigned. This piece of code also verifies if a collection of objects inserted by one peer was correctly retrieved by the others.

```
public class SimpleTest extends TestCaseImpl{
    ...
    if (!net.joinNetwork(peer, bootaddress, false, log)){
        inconclusive("I couldn't join, sorry");
    }
    ...
    @Test(testers="0",timeout=1000000, name = "tc1", step = 3)
    public void put(){
    ...
        List<PastContent> expecteds= new ArrayList<PastContent>();
        for (PastContent content : peer.getInsertedContent()) {
            log.info("Expected so far : "+content.toString());
            expecteds.add(content);
        }
    ...
    @Test(testers="*",timeout=1000000, name = "tc1", step = 4)
    public void get(){
    ...
        List<PastContent> actuals= new ArrayList<PastContent>();
        for (PastContent actual : peer.getResultSet()) {
            actuals.add(actual);
        }
        Assert.assertCollectionEquals("[Local verdict] ",expecteds, actuals);
    ...
}
```

4.3 A test case example

We present a simple test case to illustrate the elements presented in this section. This test case was wrote over the motivating example (see Example 1). It checks the correctness of the insertion of two pairs *{(1, "One"),(2, "Two")}* , by peers that join the system after its storage.

The test suite is implemented by a class named **TestSample**, which is a subclass of **TestCaseImpl**. The class **TestCaseImpl** is the implementation of the interfaces introduced into the architecture sections. The **TestSample** class contains an attribute named **peer**, an instance of the peer application. In this way, we can reach the SUT, since peer is the implementation of a SUT peer (e.g., Chord, Pastry, etc).

```
public class TestSample extends TestCaseImpl {
    private Peer peer;
    private Integer key[] = {1,2};
    private String data[] = {"One","Two"};
}
```

The objective of the method *start()* below is to initialize the peer application. Since this initialization is rather expensive, it is executed only once. The *@Before* annotation ensures that this method is performed at all peers at most one time. This annotation attributes specify that the method can be executed everywhere and that its timeout is 100 milliseconds.

```
@Before(testers="*", timeout=100)
public void start(){
    peer = new Peer(); }
```

The method **join()** asks the peer instance to join the system. The annotation *@Test* specifies that this method belongs to the test case "tc1", that it is executed by testers 0, 1 and 2, and that the execution time is measured.

```
@Test(testers="0-2", name="tc1", step=1, measure=true)
public void join(){
    peer.join ();
}
```

The method **put()** stores some data in the DHT. The annotation ensures that only tester 2 executes this method.

```
@Test(testers="2", timeout=100, name="tc1", step = 2)
public void put(){
    for(int i=0;i<2;i++){
        peer.put(key[i], data[i]);
    }
}
```

The method **joinOthers()** is similar to the method **join()**, presented above. The annotation *@Test* specifies that this method is the third action of the test case, and that only testers 3 and 4 execute it.

```
@Test(testers="3-4", name = "tc1", step=3)
public void joinOthers(){
    peer.join ();
}
```

The method **retrieve()** tries to retrieve the values stored at keys 1 and 2. If the retrieved objects correspond to the ones previously stored, the test case passes, otherwise it fails.

```
@Test(testers="3-4", name = "tc1", step=4)
public void retrieve(){
    String actual [];
    for(int i=0;i<2;i++){
        actual[i] = peer.get(key[i]);
    }

    assertEquals(data, actual);
}
```

Finally, the method **stop()** asks the peer instance to leave the system. This method is executed by all the peers.

```
@After(testers ="*",timeout=100)
public void stop(){
    peer.leave();
}
```

5 Experimental Validation

In this section, we present an experimental validation of a popular open-source DHT, FreePastry[2], an implementation of Pastry (Rowstron and Druschel [2001]) from Rice University. The objective of the experiments is to validate the feasibility of the P2P incremental testing methodology, using a code coverage criteria.

We conducted four experiments, testing FreePastry in different system settings: stable, expanding, shrinking and volatile. These experiments follow steps 1 and 2 of our methodology. The goal of the first experiment is to verify that the DHT correctly inserts and retrieves data. The goal of the second experiment is to verify whether peers that join the system after the insertion of data are able to retrieve this data, i.e., if these peers integrate correctly the system. Verify the ability of peers to reconstruct the system when several peers leave the system is the goal of the third experiment. Finally, the goal of the fourth experiment is to verify whether stable peers are able to reconstruct the system (and to retrieve the inserted data), when other peers leave and join the system.

During the experiments, we measured the code coverage to evaluate the impact of the three dimensions (functionality, scalability and volatility) on code coverage, that is, measure to which extent the quantity of inserted data, the system size and the volatility impact on the code coverage. We use a stable system composed of 16 nodes as a reference.

It has to be noticed that the paper does not focus on how to select the test cases so that they would cover all the code, which is beyond the scope of the paper. With these four typical scenarios, we want to demonstrate that volatility has an impact on code coverage (i.e., that volatility must be a parameter of a P2P test selection strategy). Additionally, we focus on volatility testing and do not test these systems on more extreme situations such as performing massive inserts and retrieves, or using

[2] http://freepastry.rice.edu/FreePastry/

very large data. Testing different aspects (e.g., concurrence, data transfer, etc.) would increase significantly the confidence on both DHTs. However, these tests were out-of-scope of this paper. They could be performed through the interface of a single peer and would not need the framework presented in this paper.

For our experiments we use two clusters of 64 machines[3] running GNU/Linux. In the first cluster, each machine has 2 Intel Xeon 2.33GHz dual-core processors. In the second cluster, each machine has 2 AMD Opteron 248 2.2GHz processors. Since we can have full control over these clusters during experimentation, our experiments are reproducible. We allocate equally one peer per cluster node. In experiments with up to 64 peers, we use only one cluster. In all experiments reported in this paper, each peer is configured to run in its own Java VM. The cost of action synchronization is negligible: the execution of an empty action on 2048 peers requires less than 3 seconds. The execution time and also the synchronization time are out-of-scope of this paper.

5.1 Test Cases Summary

In this section, we describe the test cases used to test the routing table and the DHT. Initially, we describe the test sequences that the test cases are based on. Then, we detail the test cases.

5.1.1 The routing table test sequence

In the routing table test case, testers must analyze the routing table of their peers to verify if it was correctly updated. More precisely, testers must compare the ID of peers from a routing table with the ID of peers that leave or join the system. This comparison is not trivial, because each tester only knows the ID of its peer, which is dynamically assigned. To simplify the analysis of routing tables, we use test case variables to map tester IDs to peer IDs, as shown in Section 3.6.

We implemented the test case as follows.

Name: Routing Table Test.
Objective: Test the update of the routing table.
Parameters:

- P: the set of peers that form the SUT;
- P_{init}: the initial set of peers;
- P_{in}: the set of peers that join the system during the execution;
- P_{out}: the set of peers that leave the system during the execution.

Actions:

1. System creation.
2. Volatile peers are stored in test case variables.
3. Volatility simulation.
4. Routing table verification and verdict assignment.

In the first action, a system is created and joined by all peers in P_{init}. In the second action, the IDs of P_{in} and/or P_{out} are stored in test case variables. In the third action, volatility is simulated: peers from P_{in} join the system and/or peers from P_{out} leave the system by comparing their IDs with the test case variables. In the fourth action,

[3] The clusters are part of the Grid5000 experimental platform: http://www.grid5000.fr/

each remaining peer ($p \in P_{init} + P_{in} - P_{out}$) verifies its routing table, waiting for κ seconds. Then, the routing table is analyzed whether it has references to the test case variables and a verdict is assigned. Three different test cases were written based on this test sequence:

- **Recovery from peer isolation:** The first test case consists in the departure of all peers that are present in the routing table of a given peer p. Then, we test if the routing table of p is updated within a time limit.
  As mentioned, FreePastry uses a lazy approach to update the routing table. Then, we called a *ping* method to force the update of the routing table. We executed this test twice increasing the amount of calls to the *ping* method at each time. In the first time, we called the method just once and FreePastry got an *inconclusive* verdict. Such verdict was assigned since we could not affirm that the routing table was not updated due to the laziness or to a bug. In the second time, we called the method twice within a 1 second delay, then FreePastry got a *pass* verdict.
- **Expanding system:** In the second test case, we test if the peers that join a stable system are taken into account by the older peers. To do so, we analyze the routing table of each peer that belongs to a set of peers P_{init} to test if it is correctly updated within a time limit, after the joining of a set of new peers P_{in}.
  We increased the size of the system exponentially (2^n) up to 1024 peers[4] to test the update in different system sizes. We set a maximum time to limit the test execution. We also increased this time in exponential scale (2^n), starting from 8 seconds in order to perform at least one update in the routing table. Similar to the peer isolation test, FreePastry also got a *pass* verdict. This happened because when a new peer joins a FreePastry system, it needs to communicate with all its neighbors inducing the update of their routing tables.
- **Shrinking system:** In this third test case, we test if the peers that leave a stable system are correctly removed from the routing tables of the remaining peers, within a time limit.
  We increased exponentially the size of the system and the time limit similarly to the expanding workload. FreePastry got a *pass* verdict in all executions, however, the time to get such verdict increased dramatically compared with the expanding system due to laziness. Differently from the expanding workload, a peer does not contact any neighbor when leaving the system. Then, we had to call the *ping* method to force the update of the routing table, otherwise *inconclusive* verdicts were assigned frequently.

5.1.2 The DHT test sequence

Name: DHT Test.
Objective: Test the insert/retrieve operations.
Parameters:

- P: the set of peers that form the SUT;
- P_{init}: the initial set of peers;
- P_{in}: the set of peers that join the system during the execution;
- P_{out}: the set of peers that leave the system during the execution;
- *Data* the input data, corresponding to set of pairs (key, value).

Actions:

[4] 1024 peers correspond to 8 peers per node in the clusters.

1. System creation.
2. Insertion of $Data$.
3. Volatility simulation.
4. Data retrieval and verdict assignment.

We describe the DHT test sequence as follows. In the first action, a system is created and joined by all peers in P_{init}. In the second action, a peer $p \in P_{init}$ inserts n pairs. In the third action, volatility is simulated: peers from P_{in} join the system and/or peers from P_{out} leave the system. In the fourth action, each remaining peer ($p \in P_{init} + P_{in} - P_{out}$) tries to retrieve all the inserted data, waiting for κ seconds. When the data retrieval is finished, the retrieved data is compared to the previously inserted data and a verdict is assigned. Four different test cases were written based on this test sequence:

- **Insert/Retrieve in a Stable System:** In this first test case, we configure the system to execute 4 times for different system sizes ($|P| = (16, 32, 64, 128)$). In all executions, no peer leaves or joins the system ($P_{in} = \emptyset$, $P_{out} = \emptyset$ and $P_{init} = P$). The same input data is used in all executions ($|Data| = 1,000$). The results show that FreePastry takes at least 16 seconds to get a *pass* verdict for any size of $|P|$.
- **Insert/Retrieve in an Expanding System:** In this second test case, we use a predefined number of peers ($|P| = 128$) and of input data ($|Data| = 1,000$). The test case uses different configurations, for different rates of peers joining the system. The rate is set from 10% to 50% ($|P_{init}| \times |P_{in}|$ = [(116, 12); (103,25); (90,38); (77,51); (64,64)]). No peer leaves the system ($P_{out} = \emptyset$).
  FreePastry takes at least 8 seconds to get a *pass* verdict in an expanding system for any rate of volatility. This is faster than the stable system due to Pastry's join algorithm. Whenever a new peer p joins the system it needs to find and contact a successor. Then, Pastry updates the successor list of all the impacted peers. This update floods a large portion of the system and assists the retrievals.
- **Insert/Retrieve in a Shrinking System:** In this third test case, we also use a predefined number of peers ($|P| = 128$) and of input data ($|Data| = 1,000$). Initially, all peers join the system ($P_{init} = P$). After data insertion, some peers leave the system. The rate of peers leaving the system was set from 10% to 50% ($|P_{out}|$ = (12, 25, 38, 51, 64)). No peer joins the system ($P_{in} = \emptyset$). Note that in Pastry, the data stored by a peer becomes unavailable when this peer leaves the system and remains unavailable until it comes back. Thus, in this test case, we do not expect to retrieve all data, only the remaining data is retrieved to build the verdict.
  The results show that FreePastry takes at least 16 seconds to get a *pass* verdict in a shrinking system for any rate of volatility. This is slower than the expanding one also due to Pastry's algorithm, which is lazy. The update of the successor list only happens when a peer tries to contact a successor, for instance, during retrieval.
- **Insert/Retrieve in a Volatile System:** In this fourth test case, we use the same predefined number of peers and of input data. For this test case, we define a set of stable peers $P_{stable}, P_{stable} \subset P$ and $P = P_{stable} \cup P_{in} \cup P_{out}$. The rate of stable peers was set from 90% down to 50% ($|P_{stable}|$ = (116, 103, 90, 77, 64)). The initial set of peers is composed of the stable peers and the peers that will leave the system ($P_{init} = P_{stable} \cup P_{out}$). After the data insertion, all peers from P_{out} leave the system while all peers from P_{in} join the system. FreePastry also passes this test case, for any rate of stable peers.

Name	Qualified Name	Sub-packages	Instructions	Description
Past	rice.p2p.past	3	4,606	DHT service
Transport	org.mpisws.p2p.transport	16	19,582	Transport protocol (sockets/messages)
Pastry	rice.pastry	14	26,795	Routing network (join, routing)
Replication	rice.p2p.replication	4	2,429	Object replication

Table 2 Main packages summary

5.2 Code Coverage

To analyze the impact of volatility and scalability on the different test cases presented above, we conducted several experiments, using the test case presented above, with different parameters. In these experiments, we use two Java code analysis tools for code coverage and code metrics, Emma[5] and Metrics[6], respectively.

According to these tools, FreePastry has 80,897 bytecode instructions and contains 130 packages. About 56 packages are directly concerned by the DHT implementation. The remaining packages deal with behaviors that are not relevant here: tutorials, NAT routing, unit testing, etc. In the code coverage analysis presented in this section, we focus on 4 main packages and their sub-packages, which are summarized in Table 2. These packages represent the 4 main services affected by our test cases: DHT, data transport, message routing and object replication. In all results presented here, the code coverage rate corresponds to a merge of the code covered by all peers.

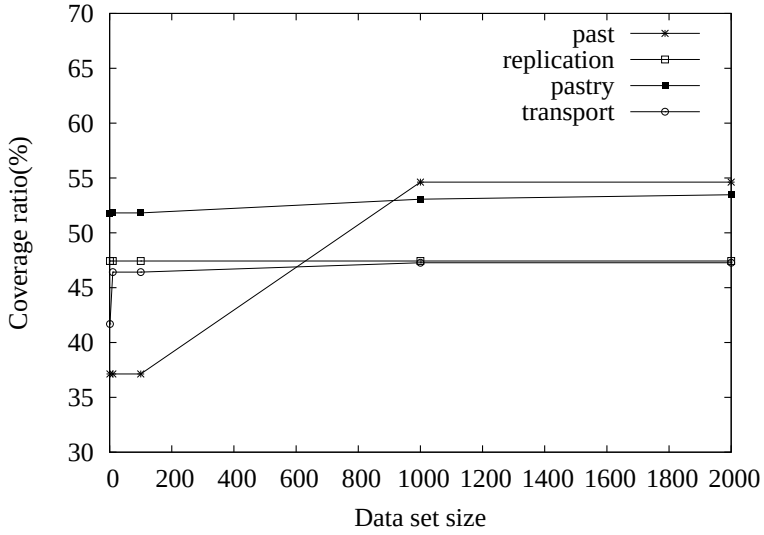


Fig. 8 Coverage on a 16-peer stable system

[5] http://emma.sourceforge.net
[6] http://metrics.sourceforge.net

For the first two experiments, we analyze the impact on the code coverage of two parameters, the size of the input data and the number of peers. As Figure 8 shows, the Past package is the most impacted by the growth of the cardinality of the input data, while the impact on the other packages is less significant. The reason for this is that the choice of the peer responsible of storing a given data depends on the data key. Thus, when a peer stores a large number of data, it must discover the responsible peers, i.e., use the **lookup()** operation. This operation will behave differently when communicating with known and unknown peers.

Figure 9 shows that the code coverage of the four packages grows when the system scales up. The explanation for this is that in small systems (e.g., 16 peers), peers know each other, and messages are not routed. When the system expands up to 128 peers, each peer only knows part of the system, making communication more complex. However, there is a limit on the coverage gains, while scaling up from 128 peers to 256 peers. Such limitation is due to some specific portions of the code (e.g., exceptions) that can be covered only by specific test cases.

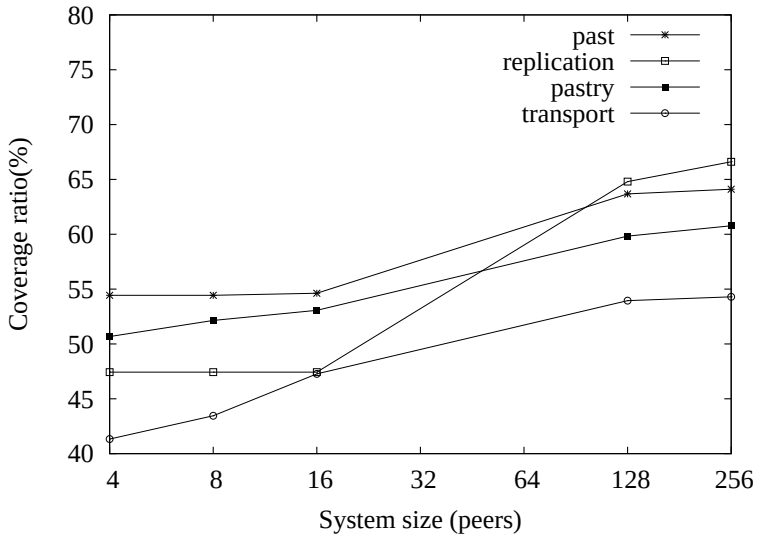


Fig. 9 Coverage inserting 1000 pairs

In the other experiments, we analyze the impact of volatility on the code coverage, using the DHT test cases. We compare these results with the coverage of the 14 original unit tests provided with FreePastry (Figure 10), which are executed locally. Figure 11 presents a synopsis of the different code coverage results. As expected, our test cases cover more code than the original unit tests, especially on packages that implement the communication protocol.

At first glance, volatility seems to have a minor impact on code coverage, since the stable test case with 256 peers yields better results than some other test cases (e.g., shrinking 128). In fact, the impact is significant because the different test cases exercises different parts of the code and are complementary. This complementarity is noticeable for the Pastry and the Past packages, where the accumulated results are

better than any other result as illustrated in Figure 10. The total accumulated coverage (Accum.+Original unit tests) shows that our tests cases and the original unit tests are also complementary.

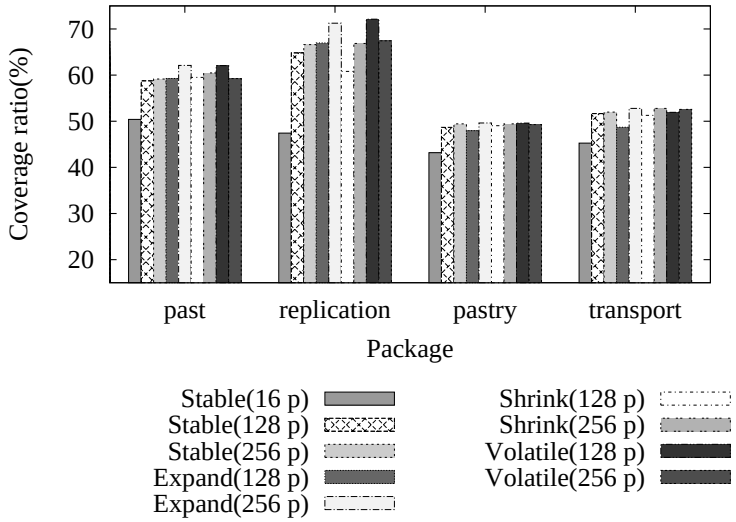


Fig. 10 Coverage by package (our test cases)

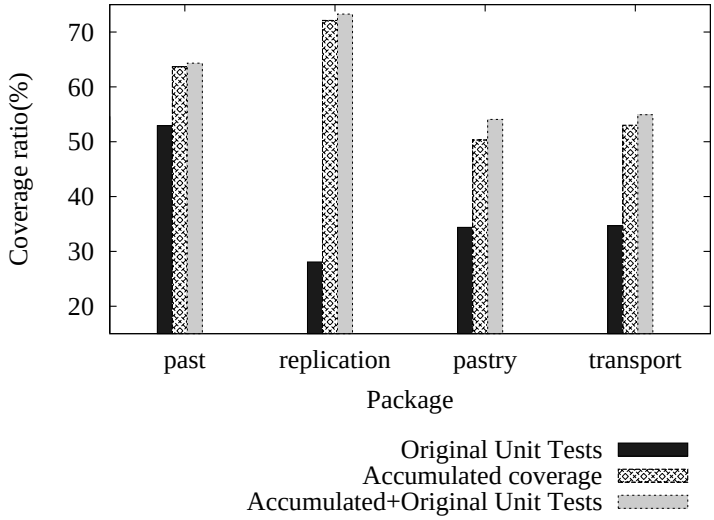


Fig. 11 Coverage by package

5.3 Learned Lessons

As expected, volatility increases code coverage. However, such increase has a limit due to some specific portions of the code (e.g., exceptions) that can be covered only by specific test cases. For instance, a test case that covers the exception threw by a look-up performed with the address of a bogus peer. This situation only happens when a peer address resides in the routing table after its volatility.

Other DHTs, such as Chord (Stoica et al. [2001]) or CAN (Ratnasamy et al. [2001]), have similar behavior to FreePastry for data storage and message routing. Therefore, a similar impact on code coverage of the size of the system and the number of data should be expected.

In spite of the test cases simplicity, the ratio of code covered by all test cases is rather important. While the impact of volatility, the number of peers and the amount of input data on the code coverage are noticeable, the only variation of these parameters is not sufficient to improve code coverage on some packages, for instance, the transport package. A possible solution to improve the coverage of these packages is to alter some execution parameters from the FreePastry configuration file. Most of the parameters deal with communication timeouts and thread delays. Yet, the number of parameters (≈ 186) may lead to an unmanageable number of test cases.

6 Related work

The main problem to build a proper P2P testing architecture is how to support the volatility of nodes without considering it as an error. Moreover, how to control the execution of tests, but avoid the inclusion of additional code inside the SUT.

The classical architecture for testing a distributed system consists of a centralized tester which sends the test inputs, controls the synchronization of the distributed system and receives the outputs (or local verdicts) of each node of the SUT.

Several solutions based on this approach have been presented. Their key feature is to distribute the execution of test cases simultaneously over a set of machines like SysUnit (McWhirter [2004]), Joshua (Kapfhammer [2001]), GridUnit (Duarte et al. [2005, 2006]), FIONA (Gerchman et al. [2005]) and BlastServer (Long and Strooper [2001]).

Figure 12 illustrates the three components of the Joshua architecture. The first component is the *TestController* which is responsible to prepare the test cases and to write them into the second component called *TestSpace*, that is a storage area. The third component, called *TestExecutor*, is responsible to consume the test cases from the *TestSpace*, to execute them, and to write the results back into the *TestSpace*. Finally, the *TestController* monitors the *TestSpace* for the results and updates the Joshua's interface with them.

The GridUnit architecture is based on Joshua. The main goal of GridUnit is to deploy and to control unit tests over a grid with minimum user intervention in order to distribute the execution of tests and speed up the testing process. To distribute the execution, each test case is transformed in a grid task. The control and scheduling of the tasks are provided by the OurGrid platform (da Silva et al. [2003]; Cirne et al. [2006]). Therefore, different test cases can be executed by different nodes. However, a single test case is only executed by a single node. Moreover, both architectures do not handle node volatility.

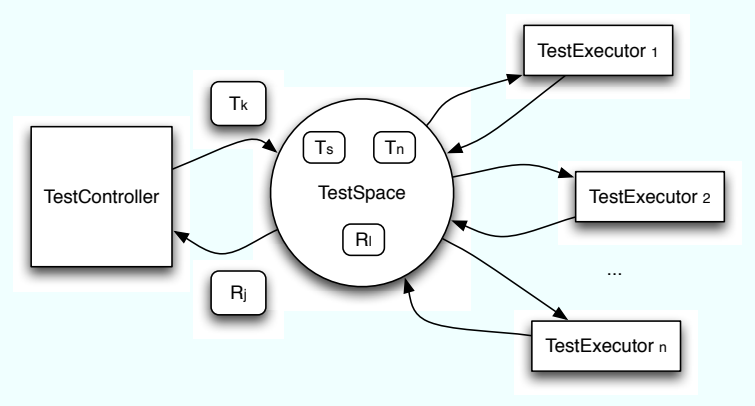


Fig. 12 The Joshua Architecture

BlastServer and FIONA are other architectures similar to Joshua. They use the client/server approach. A server component is responsible to synchronize events, while a client component provides the communication conduit between the server component and the client application. The execution of tests is based on a queue controlled by the server component. Clients requests are queued, then consumed when needed. This approach ensures that concurrent test sequences run to completion. However, they do not address the issues of network failures. Furthermore, it is not possible to decompose a test case in actions where different nodes execute different actions.

While the test case decomposition allows the execution of complex test cases, it requires actions synchronization to guarantee the correct sequence of execution. This kind of synchronization requires an architecture to coordinate when and where to dispatch the actions properly.

A testing architecture for distributed systems, called Test and Monitoring Tool (TMT) (Ulrich and Konig [1999]; Petrenko and Ulrich [2000]), decomposes test cases and coordinates them by synchronization of events. This architecture uses a global tester and a distributed tester. The global tester divides the test cases in small parts called *partial test cases* (PTC). Each PTC is assigned to a distributed tester and can be executed in parallel to another PTC with respect to a function that controls mutual exclusivity. The behavior of the distributed testers is controlled by a *Test Coordination Procedure* (TCP) which coordinates the PTCs execution by synchronization events. By using synchronization events, the distributed testers do not need to control the execution of the entire test case. Each tester runs independently its partial test case

Another testing architecture was presented for conformance, interoperability, performance and real-time testing (Walter et al. [1998]). This architecture works as a toolbox of components (e.g., communication, test coordination, etc) which can be combined to develop a specific testing architecture. Figure 13 illustrates a specific architecture for interoperability testing. Three points of communication (PC) A, B and C are monitored by their respective test components (TC) which also provide the verdicts and the control of the SUT.

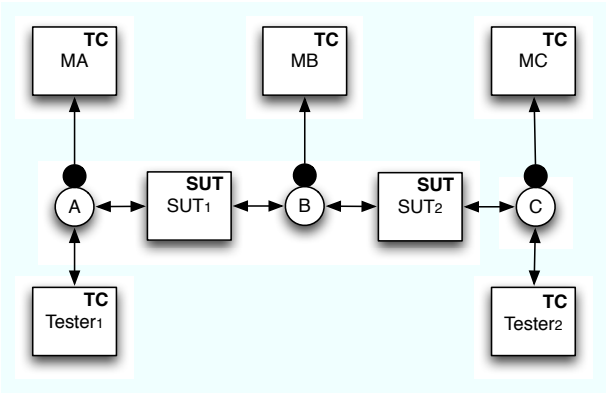


Fig. 13 An interoperability testing architecture (Walter et al. [1998])

The components can be also used to simulate failures along the tests like network delay or noise. For instance, it can be possible to create disturbs in the network to simulate volatility. However, such a disturb may be considered a failure.

The departure of nodes can also prevent the synchronization of events (Ulrich and Konig [1999]; Petrenko and Ulrich [2000]). Consequently, the whole test architecture deadlocks. This architecture has also an issue in the presence of result incompleteness, since the result sets are expected to be complete when testing traditional distributed systems.

Some other testing architectures address the volatility and scalability issues together, e.g., (Hughes et al. [2004]), P2PTester (Dragan et al. [2006]) and Pigeon (Zhou et al. [2006]). However, all of them require the inclusion of additional code in the SUT source code. This inclusion can be either manually, for instance using specific interfaces, like in P2PTester, or automatically, using reflection and aspect-oriented programming, like in Pigeon.

The inclusion of additional code is error-prone since the added code may produce errors and contaminate the test results. Furthermore, it is hard to verify if the error came from the SUT or the testing architecture.

Additionally, P2PTester and Pigeon simulate volatility using a rate (e.g., 10% of the peers leave or join the system). However, some tests can be hard or even impossible to perform in this way, because it cannot be possible to ensure the correct peers to drop/insert while testing a specific feature. For instance, if we test the routing table update, then we must control precisely which peers to drop/insert.

In our approach, we simulate volatility at a specific moment in a test scenario taking advantage of our API. For instance, a peer leaves the system after another one inserts data (i.e., Section 5.1.2). In this case, the **fail()** method is used to drop such peer.

In fact, all these architectures are used to measure the performance of P2P systems. For instance, P2PTester measures the time and costs of processing jobs such as data indexing or querying. The platform also logs the communication spawned from processing each specific query or search issued by a node (Hughes et al. [2004]).

The Pigeon framework is another platform focused on performance testing for massively multiplayer online games (MMGs). The authors believe that the performance of

MMGs is the most crucial problem that should be addressed since the network latency is impacted by the frequent propagation of updates during a game. Then, a layer called P2P Communication Model (PCM) is charged to monitor the network latency and log the exchanged messages.

Although measuring the performance of the systems can be interesting to execute both performance testing and benchmarks, it is not enough to other kinds of tests. Moreover, all of these platforms rely on logging while testing. Assuming that these platforms aim to verify correctness, they must check the log files after the test execution using an oracle. However, none of the platforms provide any oracle mechanism to do it.

There are also some model-checking tools, e.g., Bogor (Baresi et al. [2007]), SPIN (de Vries and Tretmans [2000]; Zanolin et al. [2003]) and Cadence SMV (Garlan et al. [2003]) to check a kind of P2P system called publish-subscribe system (PSS) (Carzaniga et al. [2000]). In one hand, model-checking is an attractive alternative to find bugs in systems by exploring all possible execution states (Garlan et al. [2003]; Joubert and Mateescu [2006]). In the other hand, it is still based on models and implementation problems can not be found. While they consider the volatility nature of PSS, scalability is an open issue. Along model-checking, peers are simulated as threads and the size of the SUT may be bounded by the checking machine resources. Therefore, large-scale P2P systems cannot be fully checked since implementation flaws are strongly related to such large-scale. Furthermore, none of these approaches address the volatility of peers that may lead to an exponential number of execution states.

7 Conclusion

In this paper, we presented a testing framework that considers the three dimensional aspects of P2P systems: functionality, scalability and volatility. We used this framework to conduct an extensive experimental validation using FreePastry, a popular open-source DHT, on different test scenarios.

We coupled the experiments with an analysis of code coverage, showing that the alteration of the three dimensional aspects improves code coverage, thus improving the confidence on test cases.

The next challenging issue is to propose a solution to select scenarios that guarantees the functional coverage of the P2P functions in combination with the "coverage" of volatility/scalability. Such a multidimensional coverage notion should be defined properly as an extension of existing classical coverage criteria.

As future work, we intend to move from the centralized architecture to a decentralized one. The new architecture should use distributed test controllers to have a better scalability for testing very large-scale systems. In a centralized architecture, the response time to action synchronization tends to grow linearly with the number of testers. Thus, large-scale tests could be affected by synchronization. We also intend to test other DHT implementations such as Bamboo[7] or JDHT[8].

[7] http://bamboo-dht.org/
[8] http://dks.sics.se/jdht/

References

Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, December 2004. ISSN 0360-0300.

Luciano Baresi, Carlo Ghezzi, and Luca Mottola. On accurate automatic verification of publish-subscribe architectures. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 199–208. IEEE Computer Society, 2007.

Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-80938-9.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, 1996.

Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 219–227, New York, NY, USA, 2000. ACM. ISBN 1-58113-183-6. doi: http://doi.acm.org/10.1145/343477.343622.

Kai Chen, Fan Jiang, and Chuan dong Huang. A new method of generating synchronizable test sequences that detect output-shifting faults based on multiple uio sequences. In *SAC*, pages 1791–1797, 2006.

Wen-Huei Chen and Hasan Ural. Synchronizable test sequences based on multiple uio sequences. *IEEE/ACM Trans. Netw.*, 3(2):152–157, 1995. ISSN 1063-6692. doi: http://dx.doi.org/10.1109/90.374116.

Walfredo Cirne, Francisco Brasileiro, Nazareno Andrade, Lauro Costa, Alisson Andrade, Reynaldo Novaes, and Miranda Mowbray. Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246, 2006. ISSN 1570-7873. doi: http://dx.doi.org/10.1007/s10723-006-9040-x.

Daniel Paranhos da Silva, Walfredo Cirne, and Francisco Vilar Brasileiro. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 169–180. Springer, 2003. ISBN 3-540-40788-X.

René G. de Vries and Jan Tretmans. On-the-fly conformance testing using spin. *STTT*, 2(4):382–393, 2000.

Florin Dragan, Bogdan Butnaru, Ioana Manolescu, Georges Gardarin, Nicoleta Preda, Benjamin Nguyen, Radu Pop, and Laurent Yeh. P2ptester: a tool for measuring P2P platform performance. In *BDA conference*, 2006.

Alexandre Duarte, Walfredo Cirne, Francisco Brasileiro, and Patricia Machado. Using the computational grid to speed up software testing. In *Proceedings of the 19th Brazilian Symposium on Software Engineer.*, 2005.

Alexandre Duarte, Walfredo Cirne, Francisco Brasileiro, and Patricia Machado. Gridunit: software testing on the grid. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 779–782, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-375-1.

David Garlan, Serge Khersonsky, and Jung Soo Kim. Model checking publish-subscribe systems. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software,*

10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings, volume 2648 of *Lecture Notes in Computer Science*, pages 166–180. Springer, 2003. ISBN 3-540-40117-2.

Júlio Gerchman, Gabriela Jacques-Silva, Roberto Jung Drebes, and Taisy Silva Weber. Ambiente distribuido de injeçao de falhas de comunicaçao para teste de aplicaçoes java de rede. *SBES*, 2005.

GNU Nana. URL `http://www.gnu.org/software/nana/nana.html`.

Robert M. Hierons. Testing a distributed system: generating minimal synchronised test sequences that detect output-shifting faults. *Information and Software Technology*, 43(9):551–560, 2001.

Daniel Hughes, Phil Greenwood, and Geoff Coulson. A framework for testing distributed systems. In *P2P '04: Proceedings of the Fourth International Conference on Peer-to-Peer Computing*, pages 262–263, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2156-8. doi: http://dx.doi.org/10.1109/P2P.2004.3.

Claude Jard. Principles of distribute test synthesis based on true-concurrency models. Technical report, IRISA/CNRS, 2001.

Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 2005.

Christophe Joubert and Radu Mateescu. Distributed on-the-fly model checking and test case generation. In Antti Valmari, editor, *Model Checking Software, 13th International SPIN Workshop, Vienna, Austria, March 30 - April 1, 2006, Proceedings*, volume 3925 of *Lecture Notes in Computer Science*, pages 126–145. Springer, 2006. ISBN 3-540-33102-6.

JTiger. URL `http://jtiger.org/`.

Junit. Junit website. http://www.junit.org.

Gregory M. Kapfhammer. Automatically and transparently distributing the execution of regression test suites. In *Proceedings of the 18th International Conference on Testing Computer Software*, Washington, D.C., June 2001.

Brad Long and Paul A. Strooper. A case study in testing distributed systems. In *Proceedings 3rd International Symposium on Distributed Objects and Applications (DOA'01)*, pages 20–30, 2001.

Bob McWhirter. SysUnit project, http://sysunit.codehaus.org/, 2004. URL `http://sysunit.codehaus.org/`.

Alexandre Petrenko and Andreas Ulrich. Verification and testing of concurrent systems with action races. In Hasan Ural, Robert L. Probert, and Gregor von Bochmann, editors, *Testing of Communicating Systems: Tools and Techniques, IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems (TestCom 2000), August 29 - September 1, 2000, Ottawa, Canada*, volume 176 of *IFIP Conference Proceedings*, pages 261–280. Kluwer, 2000. ISBN 0-7923-7921-7.

Simon Pickin, Clause Jard, Thierry Heuillard, Jean-Marc Jézéquel, and Philippe Desfray. A uml-integrated test description language for component testing. In *UML2001 wkshp: Practical UML-Based Rigorous Development Methods*, Lecture Notes in Informatics (LNI), pages 208–223. Bonner Köllen Verlag, October 2001.

Simon Pickin, Claude Jard, Yves Le Traon, Thierry Jéron, Jean-Marc Jézéquel, and Alain Le Guennec. System test synthesis from UML models of distributed software. *ACM - 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, 2002.

Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenkern. A scalable content-addressable network. *ACM SIGCOMM*, 2001.

David S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Software Eng.*, 21(1):19–31, 1995.

Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, Lecture Notes in Computer Science, pages 329–350. Springer, 2001.

Ina Schieferdecker, Mang Li, and Andreas Hoffmann. Conformance testing of tina service components - the ttcn/ corba gateway. In *IS&N*, pages 393–408, 1998.

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM*, 2001.

TestNG. URL `http://www.testng.org/`.

Andreas Ulrich and Hartmut Konig. Architectures for testing distributed systems. In *Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems*, pages 93–108, Deventer, The Netherlands, 1999. Kluwer, B.V. ISBN 0-7923-8581-0.

Thomas Walter, Ina Schieferdecker, and Jens Grabowski. Test architectures for distributed systems: State of the art and beyond. In Alexandre Petrenko and Nina Yevtushenko, editors, *IWTCS*, volume 131 of *IFIP Conference Proceedings*, pages 149–174. Kluwer, 1998. ISBN 0-412-84430-3.

Luca Zanolin, Carlo Ghezzi, and Luciano Baresi. An approach to model and validate publish/subscribe architectures. In *SAVCBS'03 Workshop: Proceedings of the Specification and Verification of Component-Based Systems Workshop, Helsinki, Finland, Sept. 2003.*, 2003. URL `citeseer.ist.psu.edu/zanolin03approach.html`.

Zhizhi Zhou, Hao Wang, Jin Zhou, Li Tang, Kai Li, Weibo Zheng, and Meiqi Fang. Pigeon: a framework for testing peer-to-peer massively multiplayer online games over heterogeneous network. In *Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE Volume 2, Issue , 8-10 Jan.*, pages 1028 – 1032, 2006.