A Framework for Testing Peer-to-Peer Systems

Eduardo De Almeida, Gerson Sunye, Yves Le Traon, Patrick Valduriez

A Framework for Testing Peer-to-Peer Systems

Eduardo Cunha de Almeida*
INRIA & LINA - Nantes
eduardo.almeida@univ-nantes.fr

Gerson Sunyé
LINA - Université de Nantes
gerson.sunye@univ-nantes.fr

Yves Le Traon
IT-TELECOM Bretagne
yves.letraon@telecom-bretagne.eu

Patrick Valduriez
INRIA & LINA - Nantes
patrick.valduriez@inria.fr

Abstract

Developing peer-to-peer (P2P) systems is hard because they must be deployed on a high number of nodes, which can be autonomous, refusing to answer to some requests or even unexpectedly leaving the system. Such volatility of nodes is a common behavior in P2P system and can be interpreted as fault during tests. In this paper, we propose a framework for testing P2P systems. This framework is based on the individual control of nodes, allowing test cases to precisely control the volatility of nodes during execution. We validated this framework through implementation and experimentation on an open-source P2P system.

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 [1]. 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 [6, 11]. Local to each node, test sequences or test automata can be executed, which run these 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 [17]. 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 [4, 9, 3] or Labeled Transition Systems [10, 14, 11] 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) [16, 15, 18] 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

*Partially supported by the Programme Alβan, the European Union Programme of High Level Scholarships for Latin America, scholarship no. E05D057478BR.

routed.

In this paper, we propose 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 propose 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 four test cases on a DHT 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 describes our validation through implementation and experimentation on an open-source P2P system. Section 5 discusses related work. Section 6 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 propose 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. 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.

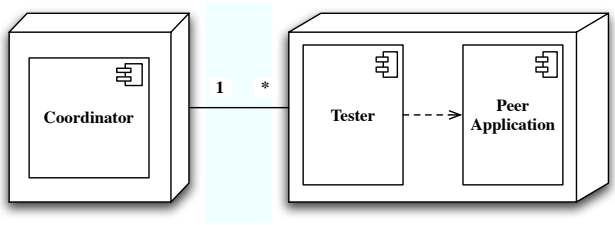

FIG. **1. Deployment Diagram**

The deployment diagram of a typical P2P system is shown in Figure 1. To test such a system, we need a framework which allows a global test to be both specified and executed. 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. 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 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 Test 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 [8]. 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. Therefore, it is crucial to combine the scalability and volatility aspects with meaningful test sequences. To test the three dimensional aspects of a P2P system, we propose the following incremental methodology:

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

Step 1 consists of conformance testing, with a minimum configuration. The goal is to provide a test sequence set (TS) efficient enough to reach a predefined test criteria. These test sequences TS must be parameterized by the number of peers, so that they can be extended for large scale testing. A test sequence is denoted by $ts(ps)$, where ps denotes a set of peers. Test sequences can also be combined to build a complex test scenario using a test language such as Tela [13].
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).
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.
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.

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).

The interface of a P2P system is spread over a network. Thus, even if all peers have exactly the same interface, testing the interface of a single peer is not sufficient to test the whole system. For instance, consider a simple test case for a DHT, where the string "one" is inserted at key 1. Then, some data is retrieved at key 1. Finally, the retrieved value is compared with the string "one" to assign a verdict to the test case. Clearly, this test case does not ensure that all peers will be able to retrieve the data stored at key 1. To address this issue, 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.

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.

Definition 1 (Distributed test case) *A distributed test case noted τ is a tuple $\tau = (A^\tau, T^\tau, L^\tau, S^\tau, V^\tau)$ 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 and V^τ is a set of variables.*

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 within 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 $a_i^\tau = (\Psi, \iota, T')$ where Ψ is a set of instructions, ι 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.

Definition 4 (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 on ι is defined as follows:*

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

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	put(14,"fourteen");
(a_3)	3,4	join();
(a_4)	3,4	data := retrieve(14);
(a_5)	3,4	assert(data = "fourteen");
(a_6)	*	leave();

This test case involves five testers $T^\tau = \{t_0 \dots t_4\}$ to control five peers $P = \{p_0 \dots p_4\}$ and six actions $A^\tau = \{a_1^\tau, ..., a_6^\tau\}$. 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 $inconclusive$.

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* [2] 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

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 (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 by two main reasons. First, to separate the execution control from testing code. Second, to simplify the deployment of the test cases: all testers receive the same test case. However, testers only execute the actions assigned to them. The available annotations are listed below:

Test. This is the main annotation, it 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.

After. Specifies that the method is executed after each test case. Its purpose is to ensure that there will be no interference among different test cases. 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

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

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_n**)**: executes a given action.
2. **leave(**P**), fail(**P**), join(**P**)**: makes a set of peers leave the system, abnormally quit or join the system.

When a tester executes a test case, it proceeds as described in Figure 2:

1. It asks the coordinator for identification, used to filter the actions that it should execute.
2. It uses java reflection to discover all actions, read their annotations and create a set of method descriptions.
3. It passes to the coordinator the set of method descriptions that it should execute and their priorities.
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.

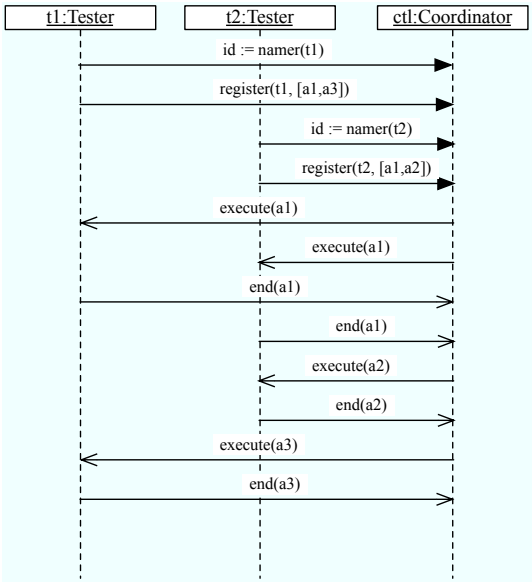

FIG. **2. Test case execution**

3.2 The Coordinator

The role of the coordinator is to control when each test action should be executed. When the test starts, the coordinator receives a set of method descriptions from each tester and associates each action to a hierarchical level. Then, it iterates over a counter representing the hierarchical level that can be executed, allowing the execution to be synchronized. From the coordinator point of view, a test suite consists of a set of actions 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.

The execution of actions follows the idea of two phase commit (2PC). In the first phase, the coordinator informs all 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:

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

3.3 Test case execution

The algorithm has three steps: 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 Example 1, tester t_2 may register the actions $A' = \{a_1, a_2, a_6\}$. Once the registration is finished, the *coordinator* builds the schedule, 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 $S(a_i^\tau)$ 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 $S^\tau(a) - T_u$, 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.

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. If the execution succeeds, then a message ok is sent to the coordinator. Otherwise, if the action timeout is reached, then a 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 string "fourteen" in a_5, 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

Algorithm 1: Test suite execution

Input: T, a set of testers; DTS, a distributed test suite
Output: $Verdict$
foreach $t \in T$ **do**
  $register(t, A^t)$;
end
foreach $\tau \in DTS$ **do**
  foreach $a \in A^\tau$ **do**
    foreach $t \in S^\tau(a)$ **do**
      send $execute(a)$ to t;
    end
    wait for an answer from all $t \in (S^\tau(a) - T_u)$;
  end
  foreach $t \in T^\tau$ **do**
    $L^\tau \leftarrow L^\tau + l_t^\tau$;
  end
  return $oracle(L^\tau, \varphi)$;
end

is also $fail$, otherwise the coordinator continues grouping each l_p^τ into L^τ. When L^τ is completed, it is analyzed to decide between verdicts $pass$ and $inconclusive$ as described in Algorithm 2. This algorithm has two inputs, a set of local verdicts (L) and an index of relaxation (φ), representing the level of acceptable $inconclusive$ verdicts. 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 2: Oracle

Input: L, a set of local verdicts; φ an index of relaxation
if $\exists l \in L, l = fail$ **then**
  return $fail$
else if $|\{l \in L : l = pass\}|/|L|) \geq \varphi$ **then**
  return $pass$
else
  return $inconclusive$
end

3.4 Writing test cases

We present below a simple test case used to test a DHT. It checks the correctness of the insertion of a single pair *(14, "fourteen")*, by peers that join the system after its storage. The test suite is implemented by a class named **TestSample**, which is a subclass of **TesterImpl**. This class contains an attribute named **peer**, an instance of the peer application.

```
public class TestSample extends TesterImpl {
  private Peer peer;
  private int key = 14;
  private String data = "fourteen"; }
```

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(place=−1, 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 on peers 0, 1 and 2, and that the execution time is measured. Note that the testers' ids are dynamically assigned by the coordinator during the registration, and are not related to the peers' ids, which are proper to the SUT.

```
@Test(from=0, to=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 peer 2 executes this method.

```
@Test(place=2, timeout=100, name="tc1", step = 2)
public void put(){
  peer.put(key, data ); }
```

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 peers 3 and 4 execute it.

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

The method **retrieve()** tries to retrieve the value stored at key 14. If the retrieved object corresponds to the one previously stored, the test case passes, otherwise it fails.

```
@Test(from=3, to=4, name = "tc1", step=4)
public void retrieve (){
  String actual = peer.get(key);
  assertEquals (data, actual ); }
```

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

```
@After(place=−1,timeout=100)
public void stop(){
  peer.leave (); }
```

4 Experimental Validation

In this section, we present an experimental validation of a popular open-source DHT, FreePastry[2], an implementation of Pastry [16] from Rice University. The experimental validation has two objectives: (i) validating the usability and efficiency of the P2P testing framework and (ii) validating the feasibility of the P2P incremental testing methodology and comparing it with classical 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. A complementary goal of the validation is to measure the impact of the three dimensions (functionality, scalability and volatility) on code coverage, that is, measure to which extent the quantity of inserted data and the system size impact on the code coverage. 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 (in other words that volatility must be a parameter of a P2P test selection strategy).

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[4]. 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. Due to the lack of space, the results of the execution time and also the synchronization time are not discussed in this paper (see [5]).

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

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

[4]The experiments are available at http://peerunit.gforge.inria.fr

4.1 Test sequence

We wrote a single test sequence to verify the implementation of the insert and the retrieval of data in FreePastry. This same test sequence is applied for different scenarios.

4.1.1 Test Sequence Summary

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: (i) System creation; (ii) Insertion of $Data$; (iii) Volatility simulation; (iv) Data retrieval; (v) Verdict assignment.

The test sequence is 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, the 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 ι seconds. When the data retrieval is finished, the retrieved data is compared to the previously inserted data and a verdict is assigned. This same test sequence is applied for four different test cases: stable system, expanding system, shrinking system and volatile system.

All test cases insert randomly generated data. The presented results are the average of several test case executions.

4.1.2 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 $|P|$.

4.1.3 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.

4.1.4 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.

4.1.5 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.

4.2 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 1. In all results presented here, the code coverage rate corresponds to a merge of the code covered by all peers.

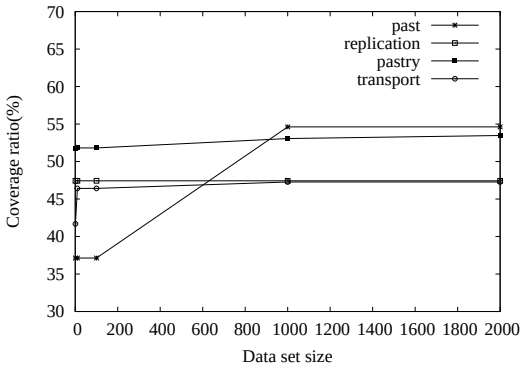

FIG. **3. Coverage on a 16-peer stable system**

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 3 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 4 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.

In the other experiments, we analyze the impact of volatility on the code coverage, using the same test cases presented in Section 4.1. We compare these results with the coverage of the 14 original unit tests provided with FreePastry (noted OT), which are executed locally. Figure 5 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., shrin-

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

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 (nodes, join, routing)
Replication	rice.p2p.replication	4	2,429	Object replication

TAB. **1. Main packages summary**

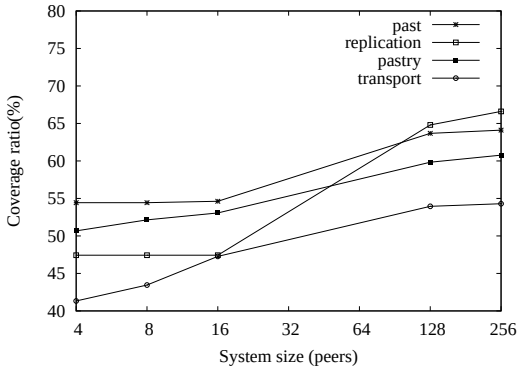

FIG. **4. Coverage inserting 1000 pairs**

king 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. The total accumulated coverage (Accum.+OT) shows that our tests cases and the original unit tests are also complementary.

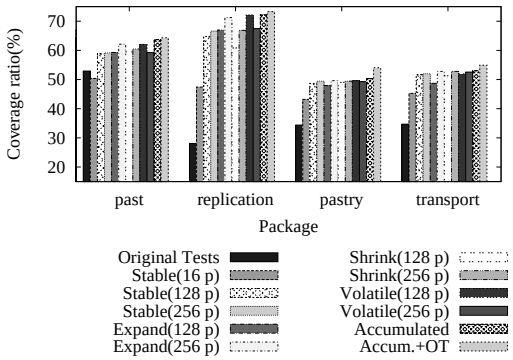

FIG. **5. Coverage by package**

4.3 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[18] or CAN[15], 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, as showed in Figure 5. 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.

5 Related work

In the context of distributed system testing, different frameworks are used to manage tests. However, all frameworks fail when dealing with the volatility of P2P systems, either interrupting the testing sequence and deadlocking the tester or assigning false-negative (i.e., false fail) verdicts.

Similar to our framework, Ulrich et al. [19] and Walter et al. [20] focus on testing distributed systems using a test controller and distributed testers. However, in their frameworks, the departure of nodes prevents the execution of synchronization events, thereby generating deadlock of the whole test architecture.

Kapfhammer [12] and Duarte et al. [8] propose a similar approach that distributes the execution of test suites on grid machines, where all machines execute the same test case at the same time. Thus, unlike our framework, it is not possible to write complex test cases where different nodes execute different actions of the same test case. Furthermore, their approach does not handle node volatility. They consider a departed node as a grid failure, which may also assign a false-negative verdict.

Butnaru et al. [7] propose a tool called P2PTester, which

measures the performance of P2P content management systems. This platform is interesting, for instance, to execute benchmarks of different DHT implementations. However, it is not adapted for testing since it does not provide an oracle.

6 Conclusion

In this paper, we proposed 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. The experimental results are reproducible and available at our web site[7].

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. During the experiments, we focused on volatility testing and did not test these systems on more extreme situations such as performing massive inserts and retrieves or using very large data. Testing different aspects (concurrence, data transfer, etc.) would increase significantly the confidence on FreePastry. However, these tests were out of the 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.

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 several test controllers to have a better scalability for testing very large-scale systems. We also intend test other DHT implementations such as Bamboo[8] or JDHT[9].

Références

[1] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, Dec. 2004.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, 1996.

[3] K. Chen, F. Jiang, and C. 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.

[4] W.-H. Chen and H. Ural. Synchronizable test sequences based on multiple uio sequences. *IEEE/ACM Trans. Netw.*, 3(2):152–157, 1995.

[5] E. C. de Almeida, G. Sunyé, and P. Valduriez. Action synchronization in p2p system testing. In *Proceedings of the 2008 International Workshop on Data Management in Peer-to-Peer Systems, DaMaP 2008*, pages 43–49. ACM, 2008.

[6] R. G. de Vries and J. Tretmans. On-the-fly conformance testing using spin. *STTT*, 2(4):382–393, 2000.

[7] F. Dragan, B. Butnaru, I. Manolescu, G. Gardarin, N. Preda, B. Nguyen, R. Pop, and L. Yeh. P2ptester: a tool for measuring P2P platform performance. In *BDA conference*, 2006.

[8] A. Duarte, W. Cirne, F. Brasileiro, and P. 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.

[9] R. 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.

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

[11] C. Jard and T. 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.

[12] G. 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.

[13] S. Pickin, C. Jard, T. Heuillard, J.-M. Jézéquel, and P. 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.

[14] S. Pickin, C. Jard, Y. Le Traon, T. Jéron, J.-M. Jézéquel, and A. 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.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenkern. A scalable content-addressable network. *ACM SIGCOMM*, 2001.

[16] A. Rowstron and P. 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.

[17] I. Schieferdecker, M. Li, and A. Hoffmann. Conformance testing of tina service components - the ttcn/ corba gateway. In *IS&N*, pages 393–408, 1998.

[18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM*, 2001.

[19] A. Ulrich and H. 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.

[20] T. Walter, I. Schieferdecker, and J. Grabowski. Test architectures for distributed systems: State of the art and beyond. In A. Petrenko and N. Yevtushenko, editors, *IWTCS*, volume 131 of *IFIP Conference Proceedings*, pages 149–174. Kluwer, 1998.

[7] http://peerunit.gforge.inria.fr/

[8] http://bamboo-dht.org/

[9] http://dks.sics.se/jdht/