Action synchronization in P2P system testing

Eduardo Cunha de Almeida, Gerson Sunye, Patrick Valduriez

To cite this version:

Eduardo Cunha de Almeida, Gerson Sunye, Patrick Valduriez. Action synchronization in P2P system testing. DAMAP 2008 - International Workshop on Data Management in Peer-to-Peer Systems (EDBT Workshop), Mar 2008, Nantes, France. pp.43-49, ⟨10.1145/1379350.1379357⟩. ⟨inria-00377411v2⟩

HAL Id: inria-00377411

https://inria.hal.science/inria-00377411v2

Submitted on 4 Jul 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Action Synchronization in P2P System Testing

Eduardo Cunha de Almeida [*]
LINA, Univ. of Nantes
eduardo.almeida@univ-nantes.fr

Gerson Suny
LINA, Univ. of Nantes
gerson.sunye@univ-nantes.fr

Patrick Valduriez
INRIA and LINA, Univ. of Nantes
patrick.valduriez@inria.fr

ABSTRACT
Testing peer-to-peer (P2P) systems is difficult because of the high numbers of nodes which can be heterogeneous and volatile. A test case may be composed of several ordered actions that may be executed on different nodes. To ensure action ordering and the correct behavior of the test case, a synchronization mechanism is required. In this paper, we propose a synchronization algorithm for executing test case actions in P2P systems. The main goal of the algorithm is to progressively dispatch the actions of a test case to a set nodes and ensure that all nodes completed the execution of an action before dispatching the next one. We validated our synchronization algorithm through implementation and experimentation on an open-source P2P system. The experimentation shows how the algorithm was able to detect implementation problems on the P2P system.

1. INTRODUCTION
P2P systems are becoming increasingly used in different kinds of applications (file sharing, video broadcast, voice over IP, etc.). But they are much more difficult to test than traditional distributed systems. They typically have very high numbers of nodes which may be autonomous (may refuse to answer) and volatile (may fail or disconnect frequently). Also, nodes tend to have similar software but very different hardware, and thus heterogeneous computing capabilities.

Although P2P systems usually have a simple public interface, the interaction between nodes is rather complex and difficult to test. For instance, distributed hash tables (DHTs) [16, 18], provide only three public operations (insert, retrieve and lookup), but need very complex interactions to ensure the persistence of data while nodes leave or join the system. Testing these three operations is rather simple. However, testing that a node correctly transfers its data to another node before leaving requires the system to be in a particular state. Setting a system into a given state requires the execution of a sequence of actions, corresponding to the public operation calls as well as the requests to join or leave the system, in a precise order.

[*]Supported by the Programme Alβan, the European Union Programme of High Level Scholarships for Latin America, scholarship no. E05D057478BR.

In a P2P system, actions can be executed in parallel, on different, heterogeneous nodes. Thus, an action can run faster or slower depending on the node computing power. Synchronization is then needed to ensure that a sequence of actions of a test case is correctly executed. For instance, suppose a simple test case where a node removes a value previously inserted by another node. In order to correctly execute this test case, the execution must ensure that the insertion is performed before the removal.

To ensure action ordering and the correct behavior of a test case, a synchronization mechanism is required. But designing such a mechanism is hard as it must scale up and deal with nodes' autonomy, heterogeneity and volatility. If the volatility is not controlled by this mechanism, then the departure of one or more peers, which may be needed by a test case, can be interpreted as an error and even interrupt the execution of the test case.

Some approaches [8, 14, 20] propose to use a test controller to synchronize the execution of test cases on a large number of nodes. However, they are not suited to P2P systems for two main reasons. First, they do not take into account the volatility of nodes and may consider as an error the normal behavior of the system and even block, waiting indefinitely for a node that already left the system. Second, they perceive the system as a set of small systems, not as a whole. The same test case is executed in all nodes, thereby limiting the accuracy of the test cases.

In this paper, we propose an action synchronization algorithm to execute test cases in P2P systems. The algorithm has two main actors: the coordinator, which is responsible to choose the actions that should be executed, and the testers, which separately control each node. Testers may invoke any public operation available on the node interface as well as make nodes leave and join the system at anytime, according to the needs of the test case. Since the coordinator only decides the actions that should be executed without actually executing them, the algorithm scales up correctly.

This paper is organized as follows. Section 2 introduces

some fundamental concepts in software testing. In Section 3, we present our synchronization algorithm. In Section 4, we describe our validation through implementation and experimentation on an open-source P2P system. Section 5 discusses related work. Section 6 concludes.

2. P2P SYSTEM TESTING

This section introduces the basic concepts of software testing which are needed to present our action synchronization algorithm in section 3.

2.1 Testing Software Systems

Model-checking and software testing are two of the most important approaches to detect software faults. Model checking [13] is a method for verifying the specification of concurrent systems. It uses temporal logic to formally specify a system as a model and symbolic algorithms to traverse this model and check if it satisfies a set of properties. Model-checking is exhaustive since the traversal algorithm completely analyzes the system specification. However, the main problem is that systems must be expressed using finite state machines, which are not adapted to express complex systems such as P2P systems that have an explosive number of states.

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: correctness, completeness, performance, security, etc. Typically, a test case is composed of a name, an intent, a sequence of input data and the expected output. Software testing complements model checking, in the sense that it verifies actual implementations and not specifications. However, testing is not exhaustive, and cannot prove that the implementation of a system is correct. The problem is then how to choose a "good" suite of test cases that can effectively verify if the system works as expected or detect its failures.

There are at least three ways for selecting the input data. The first is to perceive the system under test (SUT) as a *black-box* and use data according to its specification: public interfaces, parameter types and possibly contracts. The second, called *white-box*, analyzes the structure of the source code to generate more specific data, i.e., data that cover all the possible paths within the code. Structural analysis of code is not restricted to test case generation, and can also be used in other testing activities, such as the detection of common programming errors. The third way is to use the most common programming errors to infer the input data. These three ways are complementary since they deal with different kinds of failures.

Once a test case is executed, its results are analyzed by an *oracle*. The role of the oracle is to compare the output values with the expected ones and to assign a *verdict* to the test case. If the values are the same, the verdict is *pass*. Otherwise, the verdict is *fail*. The verdict may also be *inconclusive*, meaning that the test case output is not precise enough to satisfy the test intent and the test must be done again. There are different sorts of oracles: assertions [19], value comparison, log file analysis, manual, etc.

2.2 Testing P2P Systems

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 [6, 10, 5], Labeled Transition Systems [11, 15, 12] and uses this specification to generate a test suite that is able to verify (total or partially) whether each specified transition is correctly implemented. The tester then observes the events sent among the different nodes of the system and verifies that the sequence of events corresponds to the state machine (or the transition system). This observation can be achieved using the traces (i.e., logs) produced by each node. The integration of the traces of all nodes is used to generate an event time line for the entire system.

Distributed systems are particularly difficult to test. The system may have several possible sources of input and output, spread over the network. The nodes that compose the network may be heterogeneous, meaning that the execution time of test case actions varies for each node. Consequently, synchronization among test case actions is necessary. In the particular case of P2P systems, testing is even more complex for two main reasons. First, test cases may need to deal with node volatility and simulate the join, departure or failures of nodes at any time. Second, test cases must deal with node autonomy and deal with situations where nodes successfully execute several test-cases even though they cannot communicate with each other.

3. SYNCHRONIZATION ALGORITHM

In this section, we present our algorithm as follows. First, we define some basic concepts. Then, we describe the main steps of the algorithm to perform synchronization. Finally, we present how the algorithm handles node volatility and autonomy.

3.1 Basic Concepts

Let us denote by P the set of nodes representing the P2P system under test, T the set of *testers* used to test P (where $|T| = |P|$), by DTS the suite of test cases that test P, and by A the set of actions executed on P by DTS.

The algorithm is performed by two kinds of actors: the *testers* and the *coordinator*.A tester manages the execution of test case actions on a single node $p \in P$ and gives a local verdict at the end of a test case. The synchronization of actions is ensured by the coordinator, which also gathers local verdicts and gives the global verdict. Figure 3.1 presents the overall architecture where a coordinator synchronizes the actions of testers and each tester manages a node (i.e. a peer in a P2P system).

Definition 1 (Distributed test suite). *A distributed test suite DTS is a set of distributed test cases.*

Definition 2 (Distributed test case). *A test case noted τ is a tuple $\tau = (A^\tau, T^\tau, V^\tau, S^\tau)$ where $A^\tau \subseteq A$ is an ordered set of m actions $\{a_0, \ldots, a_m\}$, T^τ a set of n testers $\{t_0, \ldots, t_n\}$, V^τ is a set of local verdicts and S^τ is a*

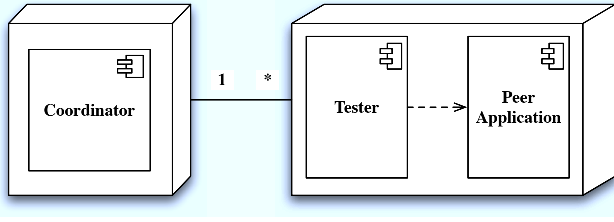

Figure 1: Deployment Diagram

schedule. Action $a_0 \in A^\tau$ is considered as the pre-condition to execute τ and a_m is considered as its post-condition.

Definition 3 (Action). *A test case action is a tuple $a_i^\tau = (\Psi^a, \theta^a, T_i^a)$ where Ψ^a is a set of instructions, θ^a is the interval of time in which a should be executed and $T_i^a \subseteq T$ is a subset of testers $\{t_0^\tau, \ldots, t_n^\tau\}$ that execute the action.*

The are three different kinds of instructions: (i) calls to the peer application public interface; (ii) calls to the tester interface and (iii) any statement in the test case programming language. The time interval θ ensures that actions do not wait eternally for a blocked peer.

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 if a node successfully resolves a given query, and continues to do so in the future.

Example 1 (Simple test case).

(a_1)	Nodes $\{p_0, p_1, p_2\}$ join the system;
(a_2)	Node p_2 inserts the string "Yonne" at key 89;
(a_3)	Pause;
(a_4)	Node p_0 retrieves data at key 89;
(a_5)	Node p_1 leaves the system;
(a_6)	Node p_0 retrieves data at key 89 again;
(a_7)	Nodes $\{p_0, p_2\}$ leave the system;
(v_0)	Tester t_0 gives a verdict;

This test case involves three testers $T^\tau = \{t_0, t_1, t_2\}$ managing seven actions $A^\tau = \{a_1, ..., a_7\}$ on three nodes $P = \{p_0, p_1, p_2\}$. The goal of the first three actions is to populate the DHT. The only local verdict is given by t_0. If the data retrieved by p_0 is the same as the one inserted by p_2, then the verdict is *pass*. If the data is not the same, the verdict is $fail$. If p_0 is not able to retrieve any data, then the verdict is *inconclusive*.

Executing τ involves two issues: (i) to perform the correct dispatching of A^τ through T^τ; (ii) to ensure that the execution of action a_i is completed before the execution of a_{i+1}.

Definition 4 (Schedule). *A schedule is a map $S^\tau = A^\tau \mapsto \Pi$, where Π is a collection of tester sets $\{T_0, \ldots, T_j\}$, where $\forall T_i \in \Pi : T_i \subseteq T$.*

S^τ maps each action to a set of testers, e.g. in example 1, $S^\tau(a_1) = \{t_0, t_1, t_2\}$.

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 types must be compatible and their values must be comparable. The local verdict v of τ on t_i is defined as follows:*

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

Once all actions of a test case τ have finished their execution, τ is able to construct its verdict V^τ. V^τ is built on the local verdicts of all testers $t \in T^\tau$. In our example, only one tester (t_0) gives a local verdict. If the data retrieved by p_0 is the same as that inserted by p_2, then the verdict is *pass*. If the data is not the same, the verdict is $fail$. If p_0 is not able to retrieve any data, then the verdict is *inconclusive*.

3.2 Algorithm

Our synchronization algorithm has three steps: registration, action execution and verdict construction. It uses only one *coordinator* to control the whole synchronization, and uses one *tester* per node to communicate with the *coordinator*.

Before the execution of a DTS, each $t \in T$ registers its actions with the *coordinator*. For instance in example 1, tester t_1 registers the actions $A^{t_1} = \{a_1, a_3, a_5\}$. Once the registration is finished, the *coordinator* maps the actions with a tester set T_i through $S^\tau(a_i)$. In our example, action a_3 is mapped to $T_3 = \{t_0, t_1, t_2\}$.

Algorithm 1: Test suite execution

```
Input: DTS, a distributed test suite
Output: Verdict
foreach t ∈ T do
  | registration(t, A^t) ;
end
foreach τ ∈ DTS do
    foreach a ∈ A^τ do
        foreach t ∈ S^τ(a) do
          | send execute(a) to t;
        end
        wait for an answer from all t ∈ (S^τ(a) − T_u) ;
    end
    foreach t ∈ T^τ do
      | V^τ ← V^τ + v_t^τ ;
    end
    return oracle(V^τ, φ) ;
end
```

The execution starts with the registration of the actions of all testers with the coordinator, which is described in algorithm 2. Then, it follows the schedule built previously to allow the gradual execution of all actions, dispatching actions to the concerned nodes. The coordinator traverses all

τ in DTS and then the actions of each τ. For each action a_i, it uses the schedule to find the set of testers T_i that are related to it, and sends the asynchronous message $execute(a)$ $\forall t \in T_i$. 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 testers controlling unavailable peers.

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 is able to start (i.e. a_2). The execution continues until the last action (i.e. a_7).

Finally, once the execution of all actions A^τ is finished, the coordinator asks all testers for a local verdict. Testers build their local verdicts by comparing their expected result E to their actual result R. If R does not contain any data, meaning that the τ execution was aborted by a timeout or by an unexpected error, the verdict is inconclusive. In our example, only tester t_0 assigns a verdict.

The registration algorithm works as follows. Initially, the *coordinator* receives a set of actions A^t from each *tester*. Then the *coordinator* reads each action a_i from A^t and builds the schedule. The registration returns 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; A^p, a set of actions
Output: id
foreach $a \in A^p$ **do**
  $S^\tau(a) \leftarrow S^\tau(a) \cup p$;
end
id++ ;
return id ;

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 θ^a is reached, then a message *error* is sent.

Algorithm 3: Action execution

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

Once each tester $t_i \in T^\tau$ gives its verdict $v^\tau_{t_i}$, the coordinator is able to produce the verdict V^τ. If any local verdict is $fail$ then V^τ is also $fail$, otherwise the coordinator continues grouping each $v^\tau_{t_i}$ into V^τ. When V^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 (V) and an index of relaxation (φ), which is further described in Section 3.4. If V contains one or more $fail$, the test case verdict is also $fail$. If the the number of *pass* over the number of local verdicts is greater than φ, then the test case verdict is *pass*. Otherwise, the verdict is *inconclusive*.

Algorithm 4: Oracle

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

The generation of the V^τ indicates that τ finished the execution, and the synchronization of actions was successful.

3.3 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 left the system.

Once 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.

3.4 Setting a global verdict

In order 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 the master 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, 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 succeeds 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 she knows how the SUT handles the completeness.

4. EXPERIMENTAL VALIDATION

In this section, we present the results of several experiments conducted using our synchronization algorithm. First, we present a performance evaluation of the algorithm, showing the response time of actions synchronization with concurrent testers. Second, we present two system tests that verify basic properties of an implementation of the Chord DHT [18].

For our experiments, we implemented our algorithm in Java (version 1.5), and we use two clusters of 64 machines[1] running 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. The implementation and tests, produced for this paper and other P2P applications, can be found in our web page.[2] We allocate the peers equally through the nodes in the clusters up to 8 peers per machine. 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 a single Java VM.

4.1 Performance of Synchronization

We first evaluate the performance of our synchronization algorithm. In order to measure the response time of action synchronization, we submitted a fake test case, composed of empty actions through a different range of testers. Then, for each action, we measured the whole execution time, which comprises remote invocations, execution of empty actions and confirmations.

The evaluation works as follows. We deploy the fake test case through several testers. The testers register their actions with the coordinator. Once the registration is finished, the coordinator executes all the test case actions inside and measures their execution time. The evaluation finishes when the execution of all actions is over.

The fake test case contains 8 empty actions (we choose this number arbitrarily) and is executed until a limit of 2048 testers running in parallel. Figure 2 presents the response time for action synchronization for a varying number of testers. The response time grows linearly with the number of nodes as expected for an algorithmic complexity of $O(n)$.

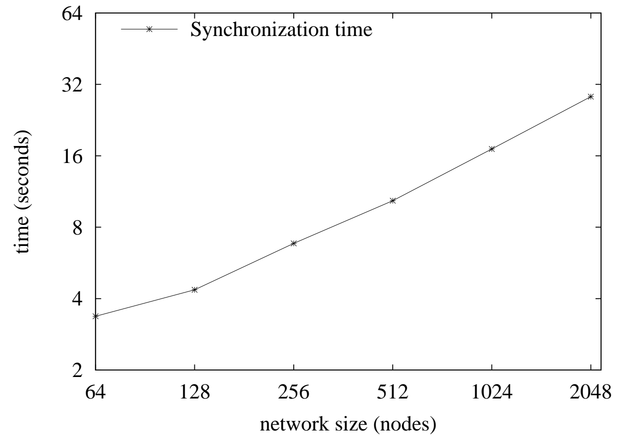

Figure 2: Synchronization algorithm evaluation

4.2 System Test

We also experimented our synchronization algorithm in to test a real P2P system. The SUT is an open source implementation of Chord [18], OpenChord [4], from Bamberg University. We performed two experiments which are compliance tests; they verify if OpenChord is a trustable implementation of Chord.

4.2.1 Testing Query Resolution

The first property verified here is the ability of resolving queries, as stated in [18]:

> *Once a node can successfully resolve a given query, it will always be able to do so in the future.*

The test contains a simple test case composed of two actions. In the first one, a node inserts some data in the DHT. In the second action, all nodes try to retrieve the data several times. The table 1 shows the test case.

Test Case Input: Expected:	<Query resolution> Integer[0:9] Integer[0:9]
(a_1)	Node p_0 inserts data;
(a_2)	All nodes retrieve data consecutively with a delay between retrievals;
(v_n)	Testers compare expected with retrieved data;

Table 1: Query Resolution test case

A delay between consecutive retrievals is set to let the SUT update its routing table during a process called stabilization, which is part of the Chord maintenance algorithm. Once the data is retrieved, it is compared to the expected output to state a verdict.

We executed this test case using a limit of 32 nodes (i.e. 8 nodes per machine). This limit was fixed to face an implementation error of OpenChord, which is further explained in the conclusion. Figure 3 presents the execution results. Starting the test with a 4 nodes network, all local verdicts are pass (i.e. all queries were resolved). However, when testing with a greater number of nodes, some verdicts are inconclusive.

In order to give a global verdict, we consider that the lack of response for some nodes is due to an implementation error since this test was executed without the volatility of peers.

4.2.2 Testing the Successor Finding

The second system test verifies if a node finds a successor within an expected time, as stated in [18]:

> *If we use a successor list of length $r = O(\log N)$ in a network that is initially stable, and then every node fails with probability 1/2, then the ex-*

[1]The clusters are part of the Grid5000 project [2]
[2]Peerunit project, http://peerunit.gforge.inria.fr

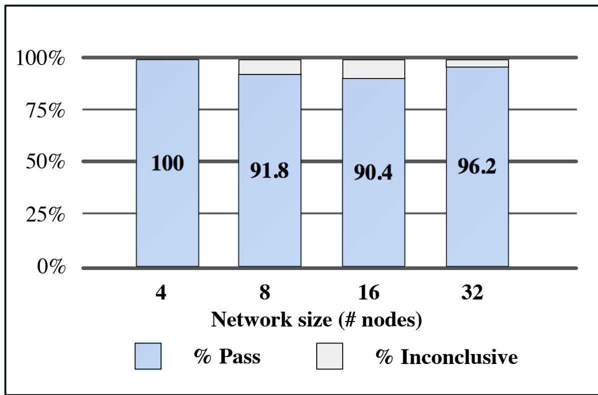

Figure 3: Query resolution evaluation

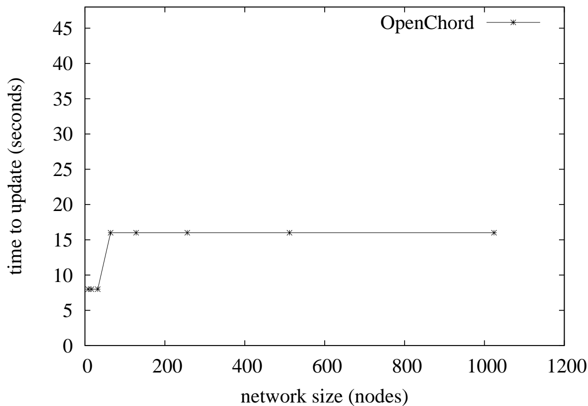

Figure 4: Time to find a successor

pected time to execute find_successor in the failed network is $O(\log N)$.

To perform this test, we developed a test case composed of two actions, the departure of 50% of the nodes and a call of the find_successor operation. Both actions are preceded by a delay for the system stabilization. The table 2 shows the test case.

Test Case	<Successor Finding>
(a_1)	50% of the nodes leave the system;
(a_2)	Live nodes call find_successor;
(v_n)	Testers verify the update of the successor list;

Table 2: Successor Finding test case

The result showed that the remaining nodes found a successor in the expected time with a network up to 1024 nodes as depicted in figure 4. Thus, this test case received a $V^\tau = pass$ for all scales of nodes.

We used a modified version of this test case to print the state of the successor list of each node after the execution of each action. We then manually analyzed the successive states of the successor lists and verified that all lists were correctly reconstructed. Although this modified version of the test case is more precise than the original one, its oracle can hardly be embedded in the test case. This is because the coordinator and the SUT do not share the same ids. More precisely, the coordinator knows its testers by an id (e.g. a,b,. . . ,z) and OpenChord assigns dynamically an id for its nodes (e.g. 1,2,. . . ,26). After the departure of nodes, the coordinator knows which testers left the system, but testers are not able to map their ids to those of OpenChord. For instance, tester 1 may know that it controls node c and that testers 15 and 18 left the system, but it does not know which ids must be deleted from the successor list.

Thus, this verification must be done after the execution. A possible solution is to use global variables for test cases (i.e. variables that are shared among testers) to construct a correspondence map during the test case execution.

5. RELATED WORK

In the context of distributed system testing, different approaches are used to manage tests, to synchronize the execution of test cases and to assign verdicts to test cases. Some of them avoid the complexity of synchronization by executing test cases entirely in a single node. However, all approaches fail when dealing with the dynamic nature of P2P systems, either interrupting the synchronization sequence and deadlocking the tester or assigning false-negative verdicts to test cases (i.e. false fail verdict).

Kapfhammer [14] describes an approach that distributes the execution of test suites. The approach is composed of three components. 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*. A solution based on this approach, called GridUnit, is presented by Duarte et al. [8, 9]. The main goal of GridUnit is to deploy and to control unit tests over a grid with minimum user intervention aiming to distribute the execution of tests to speed up the testing process. To distribute the execution, different test cases can be executed by different nodes. However, a single test case is executed only by a single node. Unlike our approach, in GridUnit, it is not possible to write more complex test cases where different nodes execute different actions of the same test case. Moreover, GridUnit does not handle node volatility. It considers a departed node as a grid failure, and this may assign a false-negative verdict to test cases.

Ulrich et al. [20] describe two test architectures for testing distributed systems using a global tester and a distributed tester. The distributed tester architecture, which is close to our algorithm, divides 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 the mutual exclusivity. The behavior of the distributed testers is controlled by a *Test Coordination Procedure*(TCP) which coordinates the PTCs execution by synchronization events. However, the departure of nodes prevents the execution of synchronization events. Consequently, the whole test architecture deadlocks.

Yet, this approach has an issue in the presence of the lack of result completeness, since the result sets are expected to be complete when testing distributed systems. Therefore, false-negative verdicts tend to be assigned to test cases when testing P2P systems. Another limitation is the execution of actions. Different nodes can execute different actions, however, the same action can not be executed in parallel by different nodes using their synchronization approach, like a_7 in the example 1. Such kind of execution can be very useful in certain kinds of tests like performance or stress testing, where several nodes insert data at the same time.

Walter et al. [21] propose a generic test architecture for conformance, interoperability, performance and real-time testing. This generic architecture works as a tool box of elements (e.g. communication, test coordination, etc), which can be combined to develop a specific testing architecture. When developing an architecture to test distributed systems, synchronization is provided by two communication elements. These elements synchronize communication among elements, however, do not synchronize test cases. If one implements such architecture to test P2P systems, the result will be something similar to GridUnit where it is not possible to write more complex tests.

Butnaru et al. [7] propose a tool called P2PTester to measure the performance of P2P content management systems. They have created a platform that measures the time and costs of processing jobs like indexing of data or querying. The platform also traces the communication spawned from processing of each specific query or search issued by a node. This platform is interesting, for instance, to execute benchmarks of different DHT implementations. However, such platform neither implements any kind of testing oracle nor synchronizes test cases.

6. CONCLUSION

In this paper, we proposed a solution to synchronize test-case actions when testing P2P systems. The solution is based on a coordinator which controls the execution of test cases, and testers which individually control each node of the system. Our solution takes into account two characteristics of P2P systems, the volatility and the autonomy of nodes. Testers can control the volatility of every system node, making them leave and join the system and ensuring that actions do not wait forever for failed nodes. Testers are also responsible for assigning local verdicts for test cases, which are analyzed by the coordinator to build a global verdict.

We also introduced an index to relax the completeness of query results. The index denotes that a certain amount of inconclusive local verdicts is acceptable when constructing the global verdict, given the autonomy of nodes that may refuse to treat some queries.

We evaluated our algorithm through three experiments. The first one shows that the coordinator scales up linearly when synchronizing up to 2048 nodes at the same time. The two other experiments test OpenChord, an implementation of the Chord DHT. Despite their simplicity, they still show two major problems of the current implementation. The first problem concerns scalability: the performance of OpenChord gets poor beyond 32 nodes. Hopefully, the next version of OpenChord will solve this problem.

The second problem concerns the retrieve operation: in some configurations (between 8 and 16 nodes) almost 10% of the retrieve operations are not able to return a result in an acceptable delay. A possible reason for this problem is that nodes organize themselves in more than one DHT (i.e. Chord ring) and the nodes from one DHT are not able to retrieve data from another DHT. In this case, testing approaches that execute the same test case in all nodes would not be able to find this problem.

Nonetheless, the experiments showed that OpenChord is robust with respect to volatility: nodes are able to rebuild their successor list, even when half of the nodes have failed.

In another extent, the experiments revealed the need of variables shared among testers. Test cases may need to use data that only becomes available during run-time and on different nodes. We intend to integrate this feature to the coordinator.

As future work, we plan to deploy test cases in a cluster to evaluate OpenChord as well as DHT implementations such as Meteor[3] or FreePastry[1] .

7. REFERENCES

[1] Freepastry, http://freepastry.rice.edu/freepastry/.

[2] Grid5000 project, http://www.grid5000.fr/.

[3] Meteor P2P distributed hash-table, JXTA project, http://meteor.jxta.org.

[4] Openchord, http://open-chord.sourceforge.net/.

[5] 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.

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

[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 *A demonstration in BDA conference*, 2006.

[8] A. Duarte, W. Cirne, F. Brasileiro, and P. Machado. Using the computational grid to speed up software testing. In *Proceedings of the 19th Brazilian Symposium on Software Engineer.*, 2005.

[9] 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.

[10] 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.

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

[12] 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.

[13] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[14] 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.

[15] 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.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenkern. A scalable content-addressable network. *ACM SIGCOMM*, 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 peertopeer lookup service for internet applications. *ACM*, 2001.

[19] Y. L. Traon, B. Baudry, and J.-M. Jézéquel. Design by contract to improve software vigilance. *IEEE Trans. Software Eng.*, 32(8):571–586, 2006.

[20] A. Ulrich, P. Zimmerer, and G. Chrobok-Diening. Test architectures for testing distributed systems. In *Proceedings of the 12th International Software Quality Week*, 1999.

[21] T. Walter, I. Schieferdecker, and J. Grabowski. Test architectures for distributed systems - state of the art and beyond, 1998.