Testing Architectures for Large Scale Systems[⋆]

Eduardo Cunha de Almeida[⋆⋆], Gerson Sunyé, and Patrick Valduriez

INRIA - University of Nantes
eduardo.almeida@univ-nantes.fr

Abstract. Typical distributed testing architectures decompose test cases in actions and dispatch them to different nodes. They use a central test controller to synchronize the action execution sequence. This architecture is not fully adapted to large scale distributed systems, since the central controller does not scale up. This paper presents two approaches to synchronize the execution of test case actions in a distributed manner. The first approach organizes the testers in a B-tree synchronizing through messages exchanged among parents and children. The second approach uses gossiping messages synchronizing through messages exchanged among consecutive testers. We compare these two approaches and discuss their advantages and drawbacks.

1 Introduction

Current Grid solutions focus on data sharing and collaboration for statically defined virtual organizations with powerful servers. They cannot be easily extended to satisfy the needs of dynamic virtual organizations such as professional communities where members contribute their own data sources, perhaps small ones but in high numbers, and may join and leave the Grid at will. In particular, current solutions require heavy organization, administration and tuning which are not appropriate for large numbers of small devices.

Peer-to-Peer (P2P) techniques which focus on scalability, dynamism, autonomy and decentralized control can be very useful to Grid data management. The synergy between P2P computing and Grid computing has been advocated to help resolve their respective deficiencies [12]. For instance, Narada [13], P-Grid [2] and Organic Grid [4] develop self-organizing and scalable Grid services using P2P interactions. The Grid4All European project [7] which aims at democratizing the Grid is also using P2P techniques. As further evidence of this trend, the Global Grid Forum has recently created the OGSA-P2P group [3] to extend OGSA for the development of P2P applications.

Grid and P2P systems are becoming key technologies for software development, but still lack an integrated solution to ensure trust in the final software, in

[⋆] Work partially funded by CAPES-COFECUB (DAAD project), CNPq-INRIA (GriData project), French ANR Massive Data (Respire project) and the European Strep Grid4All project.

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

terms of correctness and security. Although Grid and 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) [22, 24], 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. The same rationale can be applied to data grid management systems (DGMS) [15].

In Grid and P2P systems, actions can be executed in parallel, on different 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. Typical testing architectures [10, 18, 27, 9] use a central test controller to synchronize the execution of test cases on distributed nodes. This approach is not fully adapted for large scale systems, since it does not scale up while testing on a large number of nodes.

In this paper, we propose two different architectures to control the execution of test cases in distributed systems. The first architecture organizes the testers in a B-tree structure where the synchronization is performed from the root to the leaves. The second approach uses gossiping messages among testers, reducing communications among the testers responsible to execute consecutive test case actions. Since both architectures do not rely on a central coordinator they scale up correctly.

This paper is organized as follows. Section 2 discusses the related work. In Section 3, we introduce some fundamental concepts in software testing. In Section 4, we discuss the centralized approach in detail, and present two distributed approaches and their trade-off. In Section 5, we present some initial results through implementation and experimentation. Section 6 concludes.

2 Related Work

In the context of distributed systems testing, different approaches can be used either to schedule or control the execution of test case actions. However, these approaches neither ensure the correct execution of a sequence of actions, nor scale up with the system under test.

Typical Grid task scheduling techniques, using centralized [8, 21] or distributed [25] approaches are not suitable for system testing, since they do not follow the same objectives. While the objective of task scheduling is to dispatch tasks to the most available nodes, the objective of the test controller is to dispatch tasks (i.e. test case actions) to predefined nodes. More over, tasking scheduling focus on parallel execution, while the test controller must ensure the execution sequence of tasks.

In the domain of distributed system testing, Kapfhammer [18] describes an approach that distributes the execution of test cases. 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. [10, 11]. 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 failure, and this may assign a false-negative verdict to test cases.

Ulrich et al. [27] 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. Through this approach different nodes can execute different actions, however, the same action can not be executed in parallel by different nodes. 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.

3 Testing large scale distributed systems

Software testing aims at detecting faults and usually consists of executing a system with a suite of *test cases* and comparing the actual behavior (e.g. the observable outputs) with the expected one. The objective of a test case is thus both to exercise the system and to check whether an erroneous behavior occurs. The first aspect relates to test inputs (or test scenario) generation, which may be guided by various test criteria (control/based-flow based coverage criteria, specification-based coverage criteria). The second aspect concerns the way the verdict is obtained (often call the 'oracle'), which means a mechanism to check whether the execution is correct (e.g. embedded assertions).

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 [26], value comparison, log file analysis, manual, etc.

A testing technique thus includes test criteria, test cases generation techniques and mechanisms for obtaining the oracle. In this paper, we roughly define a test case as being composed of a name, an intent,a sequence of input data and the expected outputs.

Grid and P2P systems are distributed applications, and should be firstly tested using appropriate tools dedicated to distributed system testing. Distributed systems are commonly tested using conformance testing [23]. 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, 14, 5], Labeled Transition Systems [16, 20, 17] 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 nodes of the system and verifies that the sequence of events corresponds to the state machine (or to the transition system).

The classical architecture for testing a distributed system, illustrated by the UML deployment diagram presented in Figure 3, consists of a *test controller* which sends the test inputs, controls the synchronization of the distributed system and receives the outputs (or local verdicts) of each node of the system under test (SUT). In many cases, the distributed system under test is perceived as a single application and it is tested using its external functionalities, without considering its components (i.e. black-box testing). The tester in that case must interpret results which include non-determinism due since several input/outputs orderings can be considered as correct.

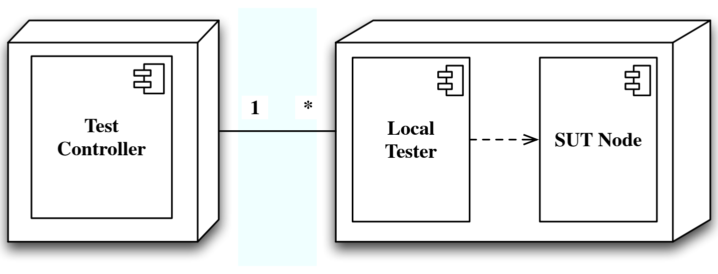

Fig. 1. Typical Centralized Tester Architecture

The observation of the outputs for a distributed system can also 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 timeline for the entire system. Most of these techniques do not deal with large scale systems, in the sense they target a small number of communicating nodes. In the case of Grid and P2P systems, the tester must observe the remote interface of peers to observe their behavior and she must deal with a potentially large number of peers. Writing test cases is then particularly difficult, because non-trivial test cases must execute actions on different peers. Consequently, synchronization among actions is necessary to control the execution sequence of the whole test case.

Analyzing the specific features of Grid and P2P system, we remark that they are distributed systems, but the existing testing techniques for distributed systems do not address the issue of synchronization when a large number of nodes are involved. Moreover, the typical centralized tester architecture can be a bottleneck when building a testing framework for these systems.

3.1 Test Case Sample

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 schedule.

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

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 Distributed Hash Table (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).

Action	**Nodes**	**Instructions**
(a_1)	0,1,2	Join the system;

(a_2)	2	Insert the string ”One” at key 1;
		Insert the string ”Two” at key 2;

(a_3)	*	Pause;
(a_4)	0	Retrieve data at key 1; Retrieve data at key 2;
(a_5)	1	Leave the system;
(a_6)	0	Retrieve data at key 1; Retrieve data at key 2;
(a_7)	0,2	Leave the system;
(v_0)	0	Calculate 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*.

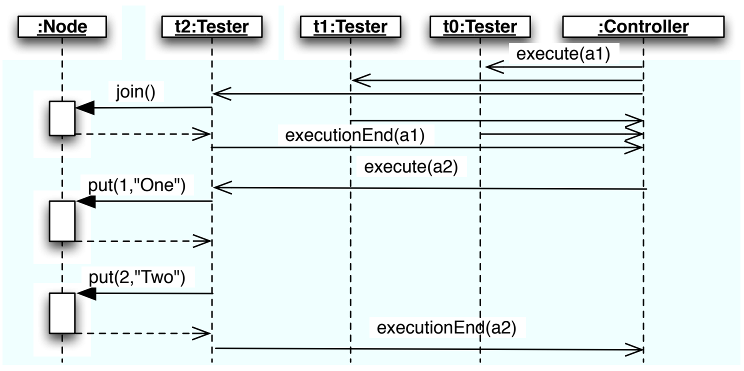

Fig. 2. Test case execution

The UML sequence diagram presented in Figure 2 illustrates the execution of the first two actions of the test case. First, the test controller asks all testers to execute action a_1. Then, each tester executes a set of instructions, interacting with the SUT. Before asking tester t_2 to execute action a_2, the test controller waits for the execution of a_1 to end. Once the execution of the test case is finished, all testers send their local verdicts to the test controller. The later compiles all local verdicts and assigns a verdict to the test case.

3.2 Problem statement

In a centralized testing architecture, the test controller dispatches actions to a variable number of testers and waits for execution results from them. The controller must then maintain a bidirectional communication channel with all testers, excluding the use of a multicasting protocol, which is fast and scalable, but unidirectional. Multicasting could be used to dispatch efficiently actions to all testers, but not to receive the execution results.

The complexity of the execution algorithm is $O(n)$, meaning that the typical architecture for testing distributed systems, using a unique test controller, is thus not adapted for testing large scale distributed systems.

4 Architecture

In this section, we present two alternatives to the centralized test controller architecture.

4.1 B-Tree

The first architecture presented here consists of organizing testers in a B-Tree structure, similarly to the overlay network used by GFS-Btree [19]. The idea is to drop the test controller and use the tester that is the root of the tree to control the execution of test cases and to assign their verdict. When executing a test case, the root dispatches actions to its child testers, who dispatch actions to their children. Once an action is executed, the leaves send their results to their parents, until the root receives all results and can dispatch the next actions.

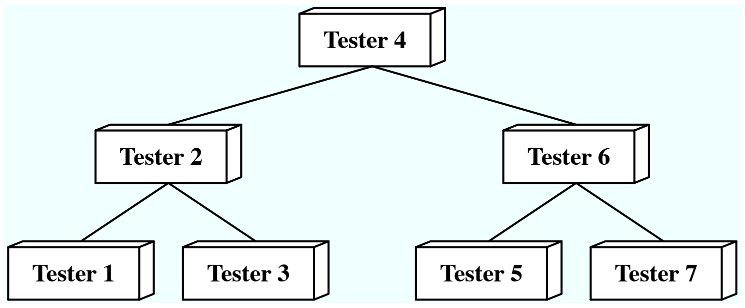

Fig. 3. B-Tree Architecture

Figure 3 presents an example of tester organisation using a B-Tree of order 1, where tester 4 is the root. Tester 4 will only communicate with testers 2 and 6. The leaves, 1, 3, 5 and 7 do not dispatch any action, they only send their results to their parents.

The order of the B-Tree is not fixed, it may vary according to the number of testers, which is known at the beginning of the execution. The goal is to have a well-proportioned tree, where the depth is equivalent to its order.

4.2 Gossiping

Besides the B-tree approach, the Gossiping is another solution to synchronize the execution of actions in a distributed manner. In Gossiping we use the same architecture used by the B-Tree approach with a tester per node, however, the synchronization of actions is executed by gossiping the coordination messages among the testers.

The Gossiping approach has the following steps. First, any node p in the system P is designated to execute the first tester t_0. This tester will act as an identifier to all the other testers t_n that join the system. The identification follows an incremental sequence from 0 up to n and is used to select the actions a node should execute. Second, t_0 creates a multicast address for each test case action. Third, the decomposed test case is deployed through P and stored at each tester. Then, each tester verifies which actions it should execute and subscribes to the suitable multicast address. Finally, the testers responsible for the first action start the execution.

A tester can play two different roles during the test case execution:

– *Busy tester*. This tester executes an action a_i and gossips its completion to the multicast address of the next action a_{i+1}. Once it has sent a gossip, it becomes an *Idle tester*.
– *Idle tester*. This tester remains idle waiting the gossips from all the *Busy testers*. Once it receives all their gossips, then it becomes a *Busy tester*.

The gossiping between these two types of testers guarantees the execution sequence of the whole test case.

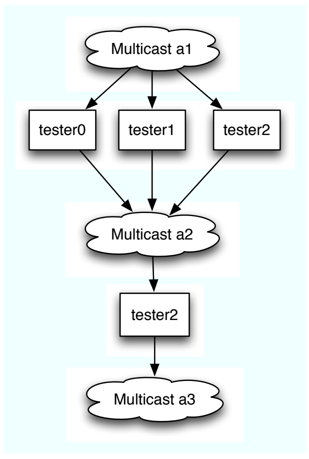

Fig. 4. Gossiping Architecture

We use the example 1 to illustrate this approach. Initially any node is chosen to be tester t_0. Then, the other nodes contact t_0 to receive an identifier n and subscribe to the suitable multicast addresses. For instance, if a tester receives $n = 1$, it subscribes to the addresses of a_1, a_3 and a_5. Figure 4 presents the first action a_1 being executed by testers $\{t_0, t_1, t_2\}$. Once the execution of a_1 is finished, the testers gossip the completion to the multicast address of the next action a_2. Once tester t_2 receives all three multicast messages, it executes a_2

gossiping in the end as well. This happens consecutively up to the last action a_7. Finally, each tester calculates a local verdict and sends it to t_0, which assigns a verdict of the entire test case.

5 Experimentation

When implementing PeerUnit [9], we have chosen a centralized architecture. This choice was due to its simplicity. However, as the performance evaluation shows, this architecture may limit the number of testers and thus, the number of nodes of the system under test. We intend to implement the two architectures presented here and evaluate their performance using the same experimentation.

For our experiments, we implemented the test controller 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 produced for this paper 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 all experiments reported in this paper, each node is configured to run in a single Java VM.

5.1 Centralized Test Controller

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 5 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 $O(n)$.

5.2 Discussion

The centralized test controller showed a linear performance in terms of response time. Although this result was expected, its implementation is easy and can

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

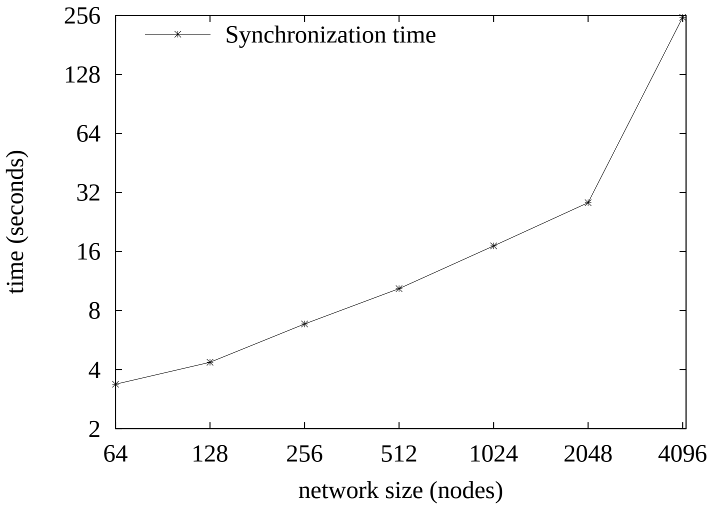

Fig. 5. Synchronization algorithm evaluation

be even used while testing in small scale environments. Our target, however, is testing in large scale environments.

The B-tree approach relies on communications between parents and children in order to reduce the communication cost and avoid the use of a centralized test controller either. In one hand, this approach scales up better than any approach described in this paper. In the other hand, it has two problems. First, a tree structure has to be built in the beginning of each execution. Second, a new action will start the execution in the root earlier than in the leaves.

The Gossiping approach can also be easy to implement and has two main advantages. First, it does not require any particular node structure (e.g. B-Tree or ring). Second, the communication can be implemented using multicast messages in order to reduce the communication cost. A weakness of this approach is the execution of consecutive actions by all the testers which requires $O(n)$ gossiping messages.

As a comparison, using the example 1 and considering that the worst case happens between actions a_2 and a_3, the B-Tree approach would need two messages to coordinate the test while the Gossiping would need three messages. In one hand, the B-Tree uses round-trip messages while the gossiping uses multicast. In the other hand, in the B-Tree a tester waits a maximum of two messages from its children while in gossiping the same tester would wait n multicast messages, $n = 3$ in this case.

6 Conclusion

In this paper, we presented two synchronization approaches to control the execution of test cases in a distributed manner using P2P techniques. Since both approaches do not rely on a central coordinator they scale up correctly.

We discuss the approaches, including the centralized, presenting their strengths and weaknesses.

We currently implement a testing tool that supports both distributed approaches for later evaluation.

References

1. Grid5000 project, http://www.grid5000.fr/.
2. Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-grid: a self-organizing structured p2p system. *SIGMOD Rec.*, 32(3):29–33, 2003.
3. Karan Bhatia. Peer-to-peer requirements on the open grid services architecture framework. OGF Informational Documents (INFO) GFD-I.049, OGSA-P2P Research Group, 2005.
4. Arjav J. Chakravarti, Gerald Baumgartner, and Mario Lauria. The organic grid: self-organizing computation on a peer-to-peer network. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 35(3):373–384, 2005.
5. 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.
6. Wen-Huei Chen and Hasan Ural. Synchronizable test sequences based on multiple uio sequences. *IEEE/ACM Trans. Netw.*, 3(2):152–157, 1995.
7. Grid4All Consortium. Grid4all: democratize the grid. World Wide Web electronic publication, 2008.
8. 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 *Euro-Par*, pages 169–180, 2003.
9. Eduardo Cunha de Almeida, Gerson Sunyé, and Patrick Valduriez. Action synchronization in p2p system testing. In *EDBT Workshops*, 2008.
10. 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.
11. 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.
12. Ian T. Foster and Adriana Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In M. Frans Kaashoek and Ion Stoica, editors, *IPTPS*, volume 2735 of *Lecture Notes in Computer Science*, pages 118–128. Springer, 2003.
13. John Hancock. The NaradaBrokering project. World Wide Web electronic publication, 2008.
14. 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.

15. Arun Jagatheesan and Arcot Rajasekar. Data grid management systems. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 683–683, New York, NY, USA, 2003. ACM.
16. Claude Jard. Principles of distribute test synthesis based on true-concurrency models. Technical report, IRISA/CNRS, 2001.
17. 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.
18. 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.
19. Qinghu Li, Jianmin Wang, and Jia-Guang Sun. Gfs-btree: A scalable peer-to-peer overlay network for lookup service. In Minglu Li, Xian-He Sun, Qianni Deng, and Jun Ni, editors, *GCC (1)*, volume 3032 of *Lecture Notes in Computer Science*, pages 340–347. Springer, 2003.
20. 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.
21. Xiao Qin and Hong Jiang. Dynamic, reliability-driven scheduling of parallel real-time jobs in heterogeneous systems. In *ICPP*, pages 113–122, 2001.
22. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenkern. A scalable content-addressable network. *ACM SIGCOMM*, 2001.
23. Ina Schieferdecker, Mang Li, and Andreas Hoffmann. Conformance testing of tina service components - the ttcn/ corba gateway. In *IS&N*, pages 393–408, 1998.
24. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peertopeer lookup service for internet applications. *ACM*, 2001.
25. Vijay Subramani, Rajkumar Kettimuthu, Srividya Srinivasan, and P. Sadayappan. Distributed job scheduling on computational grids using multiple simultaneous requests. In *11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11 2002), 23-26 July 2002, Edinburgh, Scotland, UK*, 2002.
26. Yves Le Traon, Benoit Baudry, and Jean-Marc Jézéquel. Design by contract to improve software vigilance. *IEEE Trans. Software Eng.*, 32(8):571–586, 2006.
27. Andreas Ulrich, Peter Zimmerer, and Gunther Chrobok-Diening. Test architectures for testing distributed systems. In *Proceedings of the 12th International Software Quality Week*, 1999.