Efficient Distributed Test Architectures for Large-Scale Systems*

Eduardo Cunha de Almeida[1], João Eugenio Marynowski[1], Gerson Sunyé[2], Yves Le Traon[3], and Patrick Valduriez[4]

[1] Universidade Federal do Paraná, Brazil
eduardo,jeugenio@inf.ufpr.br
[2] INRIA - University of Nantes, France
gerson.sunye@univ-nantes.fr
[3] University of Luxembourg
yves.letraon@uni.lu
[4] INRIA - LIRM, Montpellier, France
patrick.valduriez@inria.fr

Abstract. Typical testing architectures for distributed software rely on a centralized test controller, which decomposes test cases in steps and deploy them across distributed testers. The controller also guarantees the correct execution of test steps through synchronization messages. These architectures are not scalable while testing large-scale distributed systems due to the cost of synchronization management, which may increase the cost of a test and even prevent its execution. This paper presents a distributed architecture to synchronize the test execution sequence. This approach organizes the testers in a tree, where messages are exchanged among parents and children. The experimental evaluation shows that the synchronization management overhead can be reduced by several orders of magnitude. We conclude that testing architectures should scale up along with the distributed system under test.

1 Introduction

There are increasing needs of dynamic virtual organizations such as professional communities where members contribute with their own data sources and computation resources, perhaps small ones but in high numbers, and may join and leave the organization at will. In particular, current solutions require heavy organization, administration and tuning which are not appropriate for large numbers of small devices. While current Grid solutions focus on data sharing and collaboration for statically defined virtual organizations with powerful servers, Peer-to-Peer (P2P) techniques focus on scalability, dynamism, autonomy and decentralized control. Both approaches can be combined and the synergy between P2P computing and Grid computing has been advocated to help resolve their respective deficiencies [1]. Grid and P2P systems are thus becoming key

* Work partially funded by the Datluge CNPq-INRIA project.

technologies for software development, but still lack an integrated solution to validate the final software.

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) [2, 3], 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 steps, 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) [4].

Testing a large scale distributed application implies dealing with distributed test scenarios: this means synchronizing all the tasks required to execute the test scenarios and collect/aggregate test verdicts.

Typical testing architectures for distributed software rely on a centralized test controller, which decomposes test cases in steps and deploy them across distributed testers. The controller also guarantees the correct execution of test steps through synchronization messages. These architectures are not scalable while testing large-scale distributed systems due to the cost of synchronization management, which may increase the cost of a test and even prevent its execution.

From the point of view of performance, this distribution of testing tasks across the distributed architecture is intrusive and may impact the behavior of the system under test itself. This phenomenon is already known in the testing community and corresponds to the case when the observer perturbs the experience, i.e., the test environments modifies the system's behavior. A second issue occurs with error handling and diagnosis: if the same error occurs on several nodes, only one log is required while the centralized testing architecture will generate redundant logs.

The issue addressed in this paper is the optimization of the test architecture for large-scale distributed dynamic systems: the goal is to improve the performance and reduce the impact of test tasks.

This paper studies three alternatives to build a testing architecture and synchronize the test execution sequences: a centralized solution, and two distributed architectures. The distributed approach organizes the testers in a tree, where messages are exchanged among parents and children. Two strategies for building the tree are compared, depending on whether we consider the physical nodes for aggregating testers or not. The experimental evaluation shows that the synchronization management overhead can be reduced by several orders of magnitude. For the tree-based approaches, empirical results reveal the tradeoff between load balancing (deploying too many logical nodes on the same physical node degrades its performance) and thus synchronization tasks simplification. We conclude that

the distributed testing architectures scale up along with the distributed system under test, but that tuning the ideal load balancing is critical.

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

2 Related Work

A basic challenge for testing distributed systems is to synchronize the correct execution sequence of test case steps. To guarantee the correct execution at large-scale, a great deal of message exchange should be managed by some synchronization component.

The Joshua system [5] uses a centralized test controller which is responsible to prepare the test cases to be executed in a distributed manner. The BlastServer [6] system is similar to Joshua. It uses the client/server approach. A server component is responsible to synchronize events, while a client component provides the communication conduit between the server component and the client application. The execution of tests is based on a queue controlled by the server component. Clients requests are queued, then consumed when needed. This approach ensures that concurrent test sequences run to completion.

At large-scale, this approach will require a large effort from the controller to synchronize messages. For instance, one willing to stress test a DHT decides to insert a large amount of data (in the form of $\{key,value\}$) by several peers. This requires to send synchronization messages to all of these peers. Unlike our approach, the number of peers will directly impact the performance of the controller. Due to this impact, the test synchronization time will be greater than the test execution time.

A more scalable approach [7] uses a complete distributed tester architecture, which is closer to our approach. It 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 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 test steps, however, the same test step cannot be executed in parallel by different nodes.

The MRUnit system[5] is designed to unit testing of MapReduce systems (MR)[6] [8]. MR has two components: map and reduce. The map component reads a set of records (using several map instances), does a computation and

[5] MRUnit Project, http://archive.cloudera.com/docs/mrunit

[6] MR are widely used by Google and Yahoo to compute very large amounts of data, such as crawled documents, web request log, data warehousing, etc.

outputs a set of records. The reduce component consumes this output (also using several reduce instances) to group all the results together, and presents the answer to a MR computation. The MRUnit hooks a test driver to each component (the driver is analogous to the JUnit's). However, the driver only verifies the computation of the same job at maps/reduces. It does not verify complex computation where different maps compute different jobs.

The P2PTester [9] and the Pigeon [10] systems avoid the central controller to address the scalability issue. They rely on a communication layer to monitor the network and log the exchanged messages. Assuming that these platforms aim to verify correctness, they must check the log files after the test execution using an oracle approach. However, all of them require the inclusion of additional code in the SUT source code. This inclusion can be either manual, for instance using specific interfaces, like in P2PTester, or automatic, using reflection and aspect-oriented programming, like in Pigeon. The inclusion of additional code is error-prone since the added code may produce errors and contaminate the test results. Furthermore, it is hard to verify whether the error came from the system under test or the testing architecture.

3 Testing large scale distributed systems

Grid and P2P systems are distributed applications, and should be first tested using appropriate tools dedicated to distributed system testing. Distributed systems are commonly tested using conformance testing [11]. 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 [12–14], Labeled Transition Systems [15–17] 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). Note that the tester controller and the testers execute on different *logical nodes* (i.e. independent process) that may be deployed on the same *physical node* (i.e. computer device). This architecture is similar to the ISO 9646 conformance testing architecture [18]. 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 since several input/outputs orderings can be considered as correct.

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

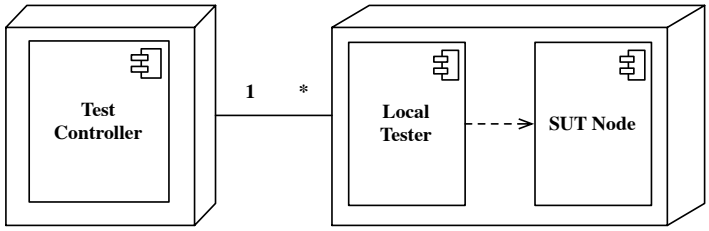

Fig. 1. Typical Centralized Tester Architecture

of these techniques do not deal with large scale systems, in the sense that 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 must deal with a potentially large number of nodes. Writing test cases is then particularly difficult, because non-trivial test cases must execute test steps on different nodes. Consequently, synchronization among test steps 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 $\tau = (S^\tau, V^\tau)$ where V^τ is a set of local verdicts and S^τ is a set of test steps. A test step is also a tuple $S = (\Psi^S, \theta^S, T^S)$ where Ψ is a set of instructions, θ is the interval of time in which the tests step should be executed and T is a set of testers that should execute this test step.

The Ψ set may contain three different kinds of instructions: (i) calls to the IUT public interface; (ii) calls to the tester interface and (iii) any statement in the test case programming language. The time interval θ sets the expected running time for Ψ and is necessary to avoid locks.

Let us illustrate these definitions with a simple distributed test case (see table 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 correctly resolves a given query that retrieves data 3 seconds after its insertion, in less than 100 milliseconds and continues to do so in the future.

This test case involves three testers $T^\tau = \{t_0, t_1, t_2\}$ managing nine steps $S^\tau = \{s_1, ..., s_9\}$ on three nodes $N = \{n_0, n_1, n_2\}$. The goal of the first three steps is to populate the DHT. The only local verdict is given by t_0. If the data retrieved by t_0 is the same as the one inserted by t_2, then the verdict is *pass*.

Table 1. Simple test case

Test Step	Testers Ψ	(Instructions)	θ
(1)	0,1,2	Join the system;	
(2)	*	Pause;	
(3)	2	Insert the string "One" at key 1;	
		Insert the string "Two" at key 2;	
(4)	*	Wait 3 sec.	
(5)	0	Retrieve data at key 1;	100 msec.
		Retrieve data at key 2;	
(6)	1	Leave the system;	
(7)	0	Retrieve data at key 1;	100 msec.
		Retrieve data at key 2;	
(8)	0,2	Leave the system;	
(9)	0	Calculate a verdict;	

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 Requirements for testing architecture

In this section, we enumerate the requirements for large-scale, distributed testing architectures.

Scalability The performance of the testing environment and specially of test step sequencing may impact the behavior of the system under test. Thus, in order to reduce the impact on the SUT, the architecture should scale at least as well as the SUT.

Test step sequencing/synchronization This requirement ensures that test steps are executed in the correct sequence and that each test step has finished in all testers before the execution of the subsequent test step. Complex test step sequencing leads to an overhead of the synchronization management.

Volatility management In some distributed systems, such as P2P, node volatility is a common behavior. The architecture must be able to simulate the entry and the departure of nodes in a fine way, without interferences to the sequencing process.

Shared variables During a test, some variables are only known dynamically and by few nodes, e.g., node ids, number of nodes, etc. The architecture should provide a mechanism that allows testers to share variables.

Error/Log handling Typical large scale systems often use the same software in several nodes, which generate similar logs. The architecture should be able to do both, reconstruct the global log timeline and detect and ignore duplicate logs.

Except for the test step sequencing, the centralized architecture does not fulfill the requirements enumerated above. The next section presents a distributed architecture that supersedes the classical centralized architecture and is able to meet these requirements.

4 Distributed Architecture

In this section, we present an alternative to the centralized testing architecture. Our proposal consists of organizing testers in a tree structure, similarly to the overlay network used by GFS-Btree [19]. The main idea is to drop the test controller and introduce the notion of *hybrid testers*, i.e., a node that is both, tester and controller.

When executing a test case, the root tester dispatches test steps to its child testers, which in turn, dispatch test steps to their children. Once a step is executed, the leaves (which are only testers), send their results to their parents, and so forth, until the root receives all results. Then, the root dispatch the next test steps in the same way.

The performance of the synchronization is highly related to the tree topology and to the distance between nodes. For instance, a high tree, i.e., with a small order, would increase the communication delay between the root and the leaves, while a wide tree, i.e., with a high order, would overload the hybrid testers.

If the distance between nodes is not taken into consideration, the communication delay between the root and the leaves may vary substantially. For instance, a path between the root and a given leaf could be composed of nodes belonging to the same physical node, while another path could be composed of intercalated nodes from two distant physical nodes.

Besides the expected impact in the synchronization performance, the distributed architecture also impacts on log handling, since hybrid testers are able to treat logs and errors before pulling them up to their parents.

In the next sections, we present two different approaches to build the tester tree. The first one builds a balanced tree, where testers are placed accordingly to their arriving time: the first nodes to connect are placed at the top of the tree. The second one introduces a more optimized structure, which avoids placing two hybrid testers at the same physical node.

4.1 Balanced Tree

In this approach, testers are organized in a tree of order m, where all nodes have at most m children and all children of a node containing less dans m children are leaves. Figure 2 presents an example of tester organization using a balanced Tree of order 2, containing 12 testers: 1 root, 6 hybrid and 5 leaf testers.

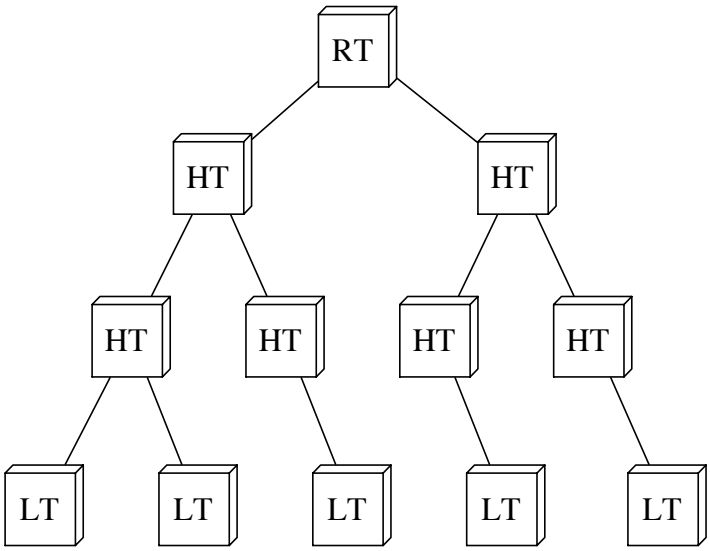

Fig. 2. Balanced Tree, 12 testers, order=2

While in theory this organization should not give good performance results since the hybrid testers are concentrated in the nearest physical nodes, in practice it does, as we will see in Section 5. This because the leaf testers, which receive the test steps in last, are placed in low-charged physical nodes, i.e., without the controller overhead. Thus, they can execute test steps quicker than other testers and reduce the global execution time.

4.2 Optimized Tree

In order to better balance the testers through the physical nodes, we added an extra constraint to the above presented tree: physical nodes contains at most one hybrid tester. Our goal was to take advantage of the context of our experiments, a grid computer with an excellent network latency. Figure 3 presents an example of tester organization using a optimized Tree of order 2, containing 9 testers: 1 root, 2 hybrid and 6 leaf testers.

Similar to the precedent tree, hybrid testers controls up to m children. However, they also control all testers belonging to the same physical node, their dependents. Contrary to the number of children, the number of dependents is not fixed, it depends on the number of testers running in the same physical node. Dependents are always leaf testers.

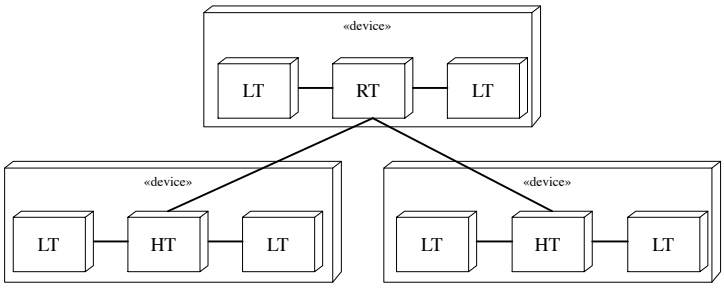

Fig. 3. Optimized Tree

5 Experimentation

In this section, we present the performance evaluation of the distributed with the centralized testing architecture. Our goal is to evaluate to which extent the distributed architecture reduces the overhead of test step synchronization management. We also evaluate the performance of the architecture in different configurations, varying the tree order, the number of testers and the number of test case steps.

All of our experiments were run on the Grid5000 platform[7] using several clusters running GNU/Linux (up to 256 nodes) connected by a 10Gbps network. We implemented our approaches in Java (version 1.5) using Remote Method Invocation (RMI) for the communication among testers. In all experiments, each tester is configured to run in a single Java virtual machine. The testers were allocated equally through the nodes up to 32 testers per physical node to obtain a large-scale testing environment. Since we can have full control over these clusters during experimentation, our experiments are reproducible. The implementation produced for this paper is open-source and can be found in our web page[8]. It was used to test two popular open-source P2P systems [20]: FreePastry and OpenChord.

5.1 Test step synchronization for up to 8,192 Testers

To measure the response time of test step synchronization, we submitted a fake test case, composed of empty test steps, across a different range of testers. Then, for each step, we measured the whole execution time, which comprises remote invocations, execution of empty test steps and confirmations. First, we verify the performance at the centralized test architecture. Then, we compare the result of the centralized with the distributed architecture.

[7] The Grid5000 platform, http://www.grid5000.fr
[8] Peerunit project, http://peerunit.gforge.inria.fr

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

The fake test case contains 8 empty test steps (we choose this number arbitrarily) and is executed until a limit of 8192 testers running in parallel. Figure 4 presents the response time for synchronization for a varying number of testers. The centralized test controller showed a linear performance in terms of response time as the number of testers increases. Although this result was expected, its implementation is straightforward and can be even used while testing in small-scale environments. Our target, however, is testing in large-scale environments.

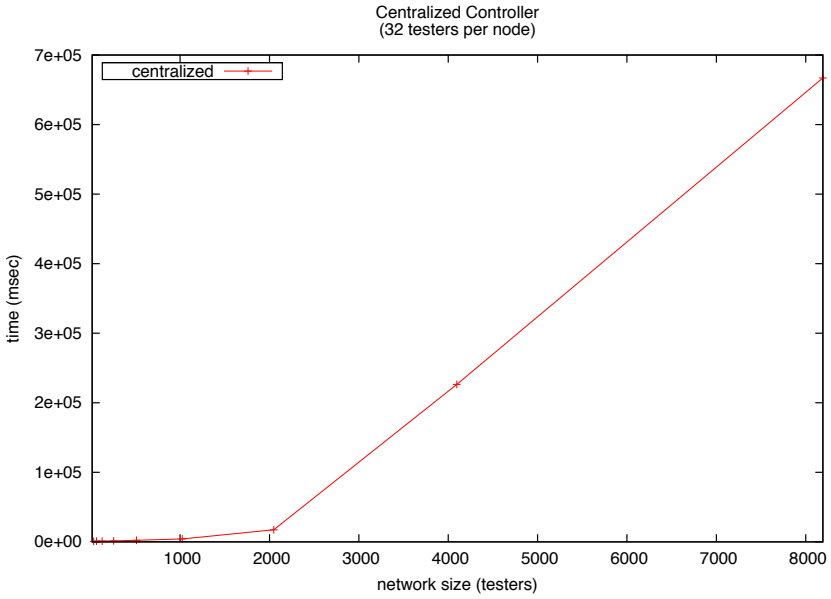

Fig. 4. Centralized Test Architecture

Figure 5 compares the response time of the centralized architecture solution with the distributed one, using both, the balanced and the optimized tree. We observe that the centralized architecture leads to an exponential increase of the overhead, which makes it unscalable. For small-scale (i.e. less than 1,024 nodes), the centralized architecture is more efficient than the distributed one. The distributed architecture, in both configurations, leads to a satisfying overhead when the number of testers increases. A disappointing phenomenon occurs, which is the very similar slopes we obtain with the two different trees. To understand the phenomenon, we study the parameters that impact the performance of these

architectures, in particular the number of logical nodes deployed per physical node and the number of test steps.

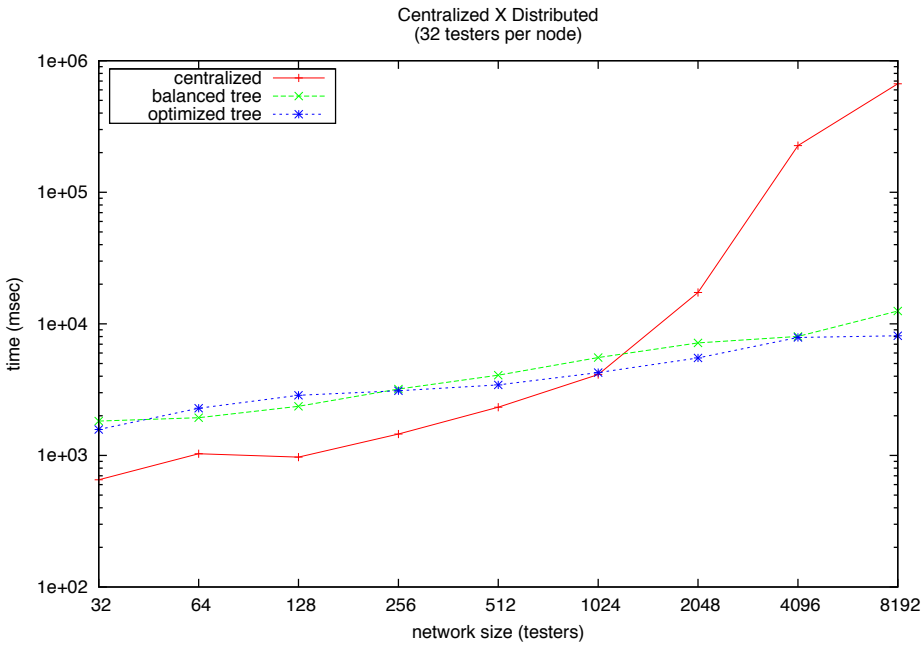

Fig. 5. Centralized X Distributed Architectures

5.2 Varying the tree order

To quantify the impact of the optimization gains on the optimized tree, we vary its order (m). The order is a parameter configured before the execution of the test case (the same test case used above, with 8 empty test steps). As expected, independently from the tree order, the time increases logarithmically.

However, as showed in Figure 6, the impact of varying the tree order is not evident. Beyond 128 testers and from $m = 2$ to $m = 8$ response time is inversely proportional to m and directly proportional to the tree height. Response time increases since the messages exchanged between the root and the leaves have a longer path among physical nodes. For instance, for 4,096 testers (256 physical nodes), the height of the tree[9] is 8 for $m = 2$ and 3 for m=8.

Still beyond 128 testers, but from $m = 8$ to $m = 64$, response time is directly proportional to m. In these cases, the overhead of controlling more testers is more

[9] Considering only hybrid testers

important than the communication overhead of heaving a higher tree, since the hight varies little. For instance, from 2 ($m = 64$, $m = 32$ and $m = 16$) to 3 ($m = 8$) for 4,096 testers.

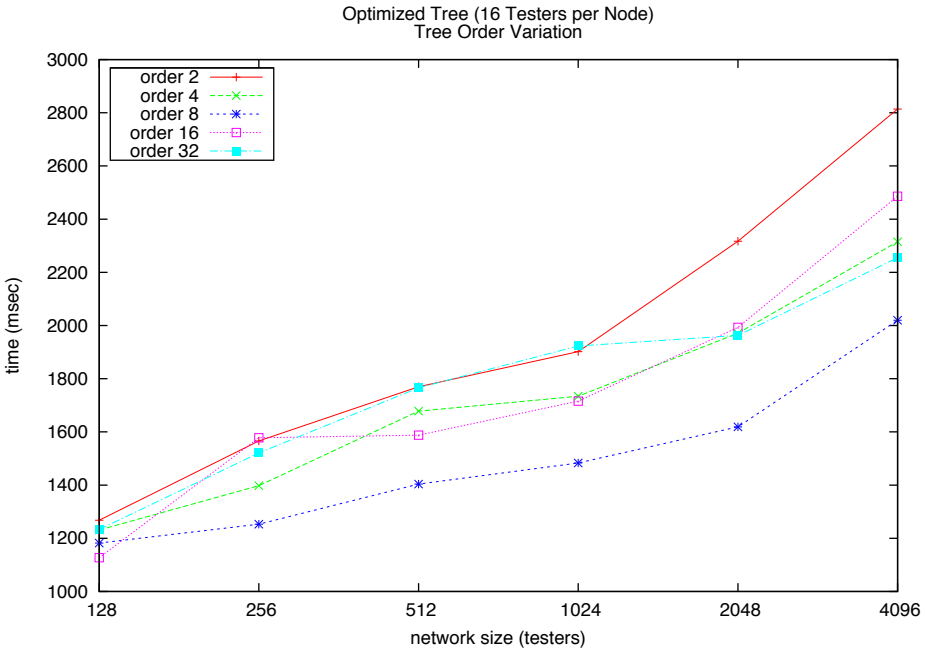

Fig. 6. Varying the tree order

5.3 Varying the number of test steps

We also investigate the impact of the number of test steps. A larger number of test steps requires a larger effort from the controller to keep the execution sequence and dispatch more steps. We limit the results to 2048 testers to easy the reading (the overhead on the central controller increases exponentially as the testing scale grows larger).

Figure 7 shows that both architectures scale up linearly, as we expected. The distributed architecture using the optimized tree yields better results in several orders of magnitude.

6 Conclusion

In this paper, we presented a distributed architecture for testing large-scale distributed systems. The architecture organizes testers in a balanced tree and

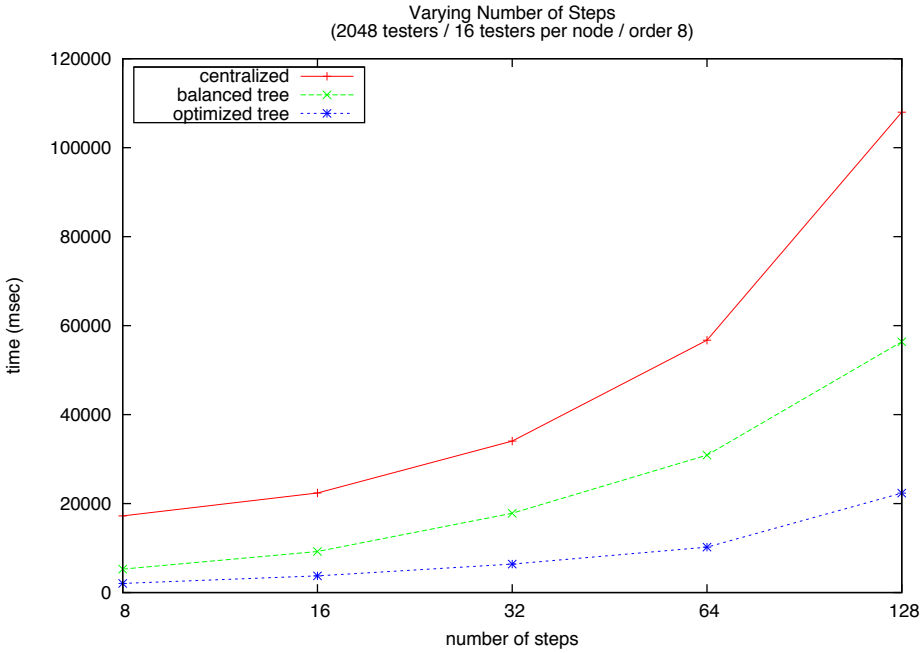

Fig. 7. Varying the number of test steps

supersedes the traditional centralized test controller by several *hybrid* testers, which are controllers and testers at the same time. After implementing the architecture, we conducted several experiments with up to 8.192 logical nodes as means to evaluate the overall performance of the architecture.

The experiments showed that our architecture scales up logarithmically, which is an important requirement for testing large-scale systems. The experiments also showed that several factors may impact the synchronization performance of the architecture: number of nodes, number of testers per node, the order of the tree, among others.

The tree order emerged as an important yet complex factor for fine-tuning the performance. The best value seems to come from an equilibrium between the height of the tree and number of testers controlled by each hybrid tester. We believe that the best tree order for a given test can be calculated before deploying the testbed.

In order to improve our architecture, we intend to implement a logging facility which is able to build a common timeline when merging logs from different nodes, and discard similar log entries.

Finally, we intend to demonstrate that false verdicts may be assigned (i.e., false-positives or false-negatives) due to low efficient testing architectures. For instance, a DHT routing mechanism, which is used to exchange messages, requires a very low time to update its entries. In large-scale testing environments,

the DHT would perform this update faster than the test steps synchronization of the centralized testing architecture. This could lead to false-positive verdict.

References

1. Foster, I.T., Iamnitchi, A.: On death, taxes, and the convergence of peer-to-peer and grid computing. In Kaashoek, M.F., Stoica, I., eds.: IPTPS. Volume 2735 of Lecture Notes in Computer Science., Springer (2003) 118–128
2. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenkern, S.: A scalable content-addressable network. ACM SIGCOMM (2001)
3. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peertopeer lookup service for internet applications. ACM (2001)
4. Jagatheesan, A., Rajasekar, A.: Data grid management systems. In: SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM (2003) 683–683
5. Kapfhammer, G.M.: Automatically and transparently distributing the execution of regression test suites. In: Proceedings of the 18th International Conference on Testing Computer Software, Washington, D.C. (2001)
6. Long, B., Strooper, P.A.: A case study in testing distributed systems. In: Proceedings 3rd International Symposium on Distributed Objects and Applications (DOA'01). (2001) 20–30
7. Ulrich, A., Zimmerer, P., Chrobok-Diening, G.: Test architectures for testing distributed systems. In: Proceedings of the 12th International Software Quality Week. (1999)
8. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI. (2004) 137–150
9. Dragan, F., Butnaru, B., Manolescu, I., Gardarin, G., Preda, N., Nguyen, B., Pop, R., Yeh, L.: P2ptester: a tool for measuring P2P platform performance. In: BDA conference. (2006)
10. Zhou, Z., Wang, H., Zhou, J., Tang, L., Li., K.: Pigeon: A framework for testing peer-to-peer massively multiplayer online games over heterogeneous network. In: 3rd IEEE Consumer Communications and Networking Conference (CCNC). (2006)
11. Schieferdecker, I., Li, M., Hoffmann, A.: Conformance testing of tina service components - the ttcn/ corba gateway. In: IS&N. (1998) 393–408
12. Chen, W.H., Ural, H.: Synchronizable test sequences based on multiple uio sequences. IEEE/ACM Trans. Netw. **3** (1995) 152–157
13. Hierons, R.M.: Testing a distributed system: generating minimal synchronised test sequences that detect output-shifting faults. Information and Software Technology **43** (2001) 551–560
14. Chen, K., Jiang, F., dong Huang, C.: A new method of generating synchronizable test sequences that detect output-shifting faults based on multiple uio sequences. In: SAC. (2006) 1791–1797
15. Jard, C.: Principles of distribute test synthesis based on true-concurrency models. Technical report, IRISA/CNRS (2001)
16. Pickin, S., Jard, C., Le Traon, Y., Jéron, T., Jézéquel, J.M., Le Guennec, A.: 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)

17. Jard, C., Jéron, T.: 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. 9646-1, I.: Conformance Testing Methodology and Framework—Part 1: General Concepts. (1994)
19. Li, Q., Wang, J., Sun, J.G.: Gfs-btree: A scalable peer-to-peer overlay network for lookup service. In Li, M., Sun, X.H., Deng, Q., Ni, J., eds.: GCC (1). Volume 3032 of Lecture Notes in Computer Science., Springer (2003) 340–347
20. de Almeida, E.C., Sunyé, G., Traon, Y.L., Valduriez, P.: A framework for testing peer-to-peer systems. In: 19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Redmond, Seattle, USA, IEEE Computer Society (2008)