# State-Driven Testing of Distributed Systems

Domenico Cotroneo, Roberto Natella, Stefano Russo, Fabio Scippacercola

Università degli Studi di Napoli Federico II
{cotroneo,roberto.natella,sterusso,fabio.scippacercola}@unina.it

**Abstract.** In distributed systems, failures are often caused by software faults that manifest themselves only when the system enters a particular, rarely occurring system state. It thus becomes important to identify these failure-prone states during testing. We propose a state-driven testing approach for distributed systems, able to execute tests in hard-to-reach states in a repeatable and accurate way. Moreover, we present the implementation and experimental evaluation of the approach in the context of a fault-tolerant flight data processing system. Experimental results confirm the feasibility of the approach, and the accuracy and reproducibility of tests.

**Keywords:** Experimental Dependability Assessment; Fault Tolerance; Fault Injection; Workload; Genetic Algorithms; State-based Testing

## 1 Introduction

Distributed computing systems are today adopted in many business- and safety-critical domains, such as air traffic control, healthcare, and e-banking systems. In these contexts, it is mandatory to perform rigorous verification and validation activities to assure that distributed systems are highly dependable.

As a matter of fact, distributed systems tend to fail in subtle ways. These failures can be caused by software faults that manifest themselves only when the system enters a particular, rarely occurring system state [1,2,3]. Failure-prone states often evade testing since they only occur for specific sets of events and inputs (*workload*), as showed in several studies on testing of distributed systems, including filesystems [4,2], DBMSs [5], and multicast and group membership protocols [2,6,7]. Thus, identifying these states during testing is a challenging problem. This problem is exacerbated by the non-determinism of distributed systems, the need for minimal instrumentation of the system, and the presence of Off-The-Shelf (OTS) components whose internals are unknown. Past studies have mainly focused on exercising the system using synthetically generated workloads [8,9], or using workloads derived from performance benchmarks [5,10,11]. Nevertheless, these approaches are not effective at covering rare (*hard-to-reach*) states. Other approaches generate a workload from stochastic or non-deterministic models of the system, but do not scale well for complex systems [12,13].

In our previous paper [14], we proposed a workload generation technique that automatically drives the system's state towards the hard-to-reach states. In this

paper, we integrate this technique into a state-driven testing approach able to execute tests in hard-to-reach states, in a repeatable and accurate way. It does not rely on a detailed model of the system in terms of probabilities or time, and is suitable for testing the *actual implementation* of complex, OTS-based, distributed systems. Moreover, we present the implementation and experimental evaluation of the approach in the context of a fault-tolerant flight data processing system. The evaluation shows the feasibility of the approach and its ability to perform accurate and reproducible tests in the correct global state.

The paper is organized as follows. Section 2 presents past work on state-driven testing of distributed systems. Section 3 provides basic concepts and assumptions, and Section 4 describes our approach. Section 5 and 6 presents the experimental evaluation. Section 7 closes the paper.

## 2 Related work

Studies on the verification of distributed systems can be classified into two broad classes: analytical-simulation studies and experimental ones. Experimental studies, in which our work is included, assess the actual implementation of a system by executing it. They include, for instance, fault injection methods, which assess fault tolerance mechanisms and algorithms through the deliberate injection of faults in the actual system or in a prototype [6].

In experimental studies, *model-based testing* (MBT) approaches are commonly adopted for generating test cases from a formal description of the system [15]. For instance, *conformance testing* and *FSM-based testing* approaches generate test cases aimed at covering the states of the model and at assuring that the system evolves as described by the model. Early approaches adopted graph-searching techniques to identify inputs able to drive the system along a given path in the state model [16]. Later approaches [12,13] extended these approaches to drive the system state in spite of random factors that change the system state in unpredictable ways. Nevertheless, the application of these approaches in complex systems is limited by scalability issues due to state space explosion, the need for a detailed model of the system, and by restrictive assumptions they implicitly make about the system: for instance, they only consider “stable” states, in which the system waits for inputs or events [17].

For these reasons, fault injection approaches do not rely on a system model to generate a workload. Some of them assess dependability by adopting a workload *representative* of the real workload that will be experienced during operation [5,10,11], in a similar way to performance benchmarks. In other cases, synthetic workloads are randomly generated, according to a high-level workload specification provided by the tester [18], for instance in terms of input probability distributions [8,19]. Moreover, most fault injection studies randomly inject faults during an experiment, repeating this process several times [20,21,7]. In these studies, the tested system states are limited to those exercised by the considered workload, and testers must manually tune the workload in order to bring the system in “hard-to-reach” states, from which they can perform tests. Moreover,

random injection can require a significant number of experiments to "hit" the system at hard-to-reach states.

More advanced fault injection approaches trigger the injection when a specific events occur in the system [22,23,2], and perform an a-posteriori state-based sampling of experiments to compute dependability measures [24,2]. For instance, Loki [2] considers the global state of a distributed system for triggering fault injection: to assure that a fault has been injected in a desired state, it performs an off-line analysis of execution traces and repeats the experiment if the injection has been triggered in a wrong state. These approaches still rely on a workload provided by the tester, either hand-written or using a representative workload, which does not assure that all important states are covered during testing. Compared to these works, our approach actively tunes the workload in order to cover a specific state specified by the tester, thus complementing experimental assessment approaches such as Loki. In our preliminary work [14], we discussed the issues behind state-driven workload generation in distributed systems, and first proposed the use of genetic algorithms to this aim. In this study, we integrate this technique in a comprehensive approach for fault injection testing.

## 3 Basic concepts and problem statement

In the design of our approach for state-driven testing, we make practical assumptions about the architecture of the distributed system (DS) under evaluation. We consider DSs in which a set of *services* is exported by a *frontend* process, masking the complexity of the system to its users (Fig. 1). A client sends requests to the frontend process by means of one or more messages, the frontend interacts with the other processes of the DS and, once the computation has finished, replies to the client. In this context, a *workload* $W$ consists of a set of service requests generated during an execution. This view of DSs applies to several systems, including orchestrated web services and three-tier web applications.

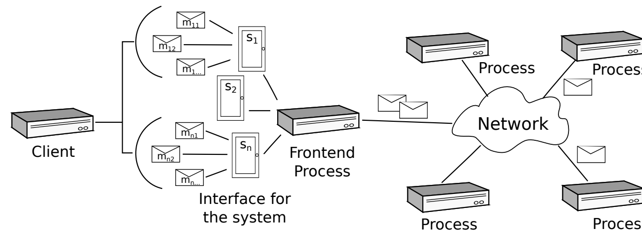


**Fig. 1.** The distributed system under test.

In state-based testing of DSs, the workload is adopted to bring the system in a global state defined by the tester (*target state*), in order to let him to perform a test right after the DS has reached the target state, for instance by submitting a set of inputs or by injecting a fault while the system is in the target state. The aim of *State-Driven Workload Generation* (SDWG) is to search for a workload

$\overline{W}$ such that the likelihood that the system under test (SUT) spends at least a period $\tau$ in the target state is high enough to allow the accurate and reproducible test execution in the desired state. The $\tau$ period includes the time for allowing a *Test Executor* (TE) program to notice (by collecting event logs) that the DS has entered the target state during the experiment, and to perform the test after the state has been reached: for instance, in a fault injection experiment, the TE (e.g., a fault injection tool) will require a small amount of time to corrupt a message or to kill a process [22,2].

The state of an individual process in the DS is referred to as *local state*, whereas the *global state* of the DS, denoted with $s \in \mathcal{S}$, is the union of all the local states. The target state or, more in general, the *set of target states* $\mathcal{S}_G$, is a subset of the global states in which the tester aims to perform a state-driven test. Local and global states of the DS, and target states, should be defined by the tester before generating state-driven tests. We refer as the system model to a high-level specification (using a formalism such as Finite State Machines (FSM) or Petri Nets (PN)) by which the tester describes the set of global states, including the target states. The tester should define the system model on the basis of system requirements and its high-level design. The model should account for the state of local resources and the state of computation at each process, in addition to the testing goals. The model can be specified using well-known formalisms such as Finite State Machines (FSM) or Petri Nets (PN). Using the system model, the tester can focus workload generation on those target states that are important for testing. For instance, to test the effectiveness of a deadlock detection mechanism in a distributed DBMS, the system model should reflect the contents of the lock table and distributed transactions. The target state can be expressed in terms of markings of a PN, e.g., in terms of number of tokens in places that represent the ongoing execution of a transaction. More detailed examples of high-level system models adopted for fault injection testing of a distributed filesystem and a group membership protocol are provided respectively in [4,25].

For SDWG, we only require a relatively simple model that reflects the software under development at a high-level of abstraction, which should not necessarily provide details about low-level hardware and software layers of the system (e.g., OS, middleware). In particular, we do not require the system model to characterize the *time* and the *probability* of events in the system, but only the *relationship* between events and states: time, including communication and computation delays, can be unfeasible to characterize even in a probabilistic way, especially for complex distributed systems with third-party and OTS components, whose internals are unknown. Since the time of events are unknown, transitions in the system model are not timed, and only express the relationship between events and the state of the DS (according to [15], it is an *untimed*, *non-deterministic* and *operational* model). In our approach, the system model is used *after* the execution of the workload to obtain, from raw event logs of an execution, the sequence of states that the system has followed during the execution, and refine the workload based on this feedback.

## 4 A State-driven testing approach

We are proposing an MBT approach composed by two phases, namely the *workload search phase* and the *testing phase*. The workload search phase is only briefly summarized in the following because it has been the focus of our previous work [14], while the testing phase is the main topic of this section. Fig. 2 shows the overall approach: the tester first *searches a workload*, using the *Workload Generator* (WG) [14], then performs the actual *testing* of the SUT using the workload $\overline{W}$ found in the previous phase. To find $\overline{W}$, the WG applies "candidate" workloads to the system, and evaluates whether such workloads bring the system in the target state. The WG determines if the system reached the target state collecting the system events during the execution, e.g., messages and outputs produced by processes, and analyzing them after the events have been "translated" into the history of global states traversed by the system.

In the testing phase, the tester links his module, the *Test Executor*, to the WG: the TE is responsible for executing tests, e.g., it may be deputed to inject a fault and to observe its effects on the SUT. The WG supplies again the workload $\overline{W}$ to the system, but here, during runtime, it triggers the Text Executor when it notices the occurrence of *test triggering conditions* (e.g., a specific sequence of messages sent within the system). These conditions are defined by the WG such that the likelihood that the test is performed in the target global state is maximized. This likelihood represents an evaluation of the accuracy and reproducibility of the test when using a given workload. Test reproducibility allows their re-execution after applying a fix, given that the fix does not impact the system model or the execution of $\overline{W}$. The likelihood is evaluated by the WG during the workload search phase, so the search can be stopped when it is high enough. Moreover, after the execution of a test, the WG framework checks whether the experiment has been conducted in the correct global state, in order to assure the correctness of results. The test is repeated in the unlikely case that the state of the test was not the desired one.

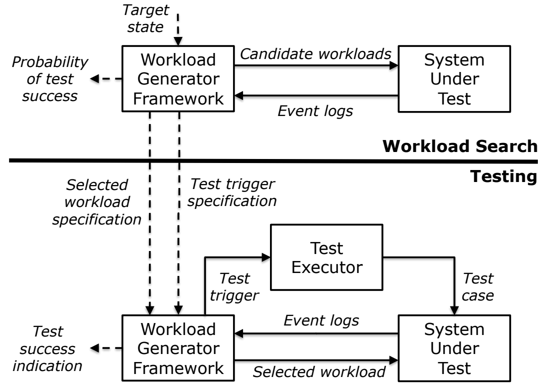


**Fig. 2.** Workflow of a test using the proposed approach.

### 4.1 The workload search phase

In the workload search phase (described in detail in [14]), the WG interacts with the DS under test in a closed-loop configuration. It exercises the DS with a workload, analyzes its behavior, and modifies the workload until a specified target state is reached. In this loop, the WG alternates an *on-line* phase, in which the DS is executed, and an *off-line* phase, in which the behavior of the DS is analyzed. The distinction between the off-line and on-line phases allows to reduce the intrusiveness of the WG, since only minimal information is collected during the on-line phase, and most of the processing for analyzing the system evolution and computing the workload occurs off-line. In the on-line phase, the WG executes for a fixed time period the DS with a candidate workload $W$. Then, in the off-line phase, the WG analyzes the behavior of the system through *event logs* collected during execution, and evaluates whether the target state has been reached. The off-line phase adopts a *Petri Net system model* (Section 3) to obtain, from raw event logs of an execution, the sequence of states that the system reached during the execution, and how much time has been spent at each state. Candidate workloads are iteratively generated and executed until the target state is reached with a given probability and for a given sojourn time.

Local events are collected at each process of the DS, timestamping them using local clocks. When the experiment is over, an *off-line synchronization algorithm* is executed to align the events of an execution on a single *global timeline* [2,26]. Off-line synchronization was preferred over on-line synchronization approaches, such as NTP, since on-line synchronization protocols exchange packets during the execution of the system and can thus interfere with its evolution. For each event, the algorithm estimates a *lower* and an *upper bound* of its timestamp, representing the *uncertainty interval* of the event. We showed in [27] that, when a PN system model is adopted, the global state of the system can be exactly identified in those periods where uncertainty intervals do not overlap.

A workload $W$ leads the SUT to traverse one or more global states $s_k \in \mathcal{S}$, and sojourns in each of them for a finite time $d_k > 0$. The behavior of the SUT under a workload $W$ is evaluated from a set of executions. The sequence of all the states traversed by the system in an execution under the workload $W$ forms an *execution report* $r_i \in \mathcal{R}_W$, where each state traversal is denoted with $(s_k, d_k)$.

The WG adopts a *workload configuration* $w_c \in W_c$ to represent workloads; $w_c$ is a vector of parameters, representing the frequency and the type of requests to be sent, i.e., the workload to be generated. The tester should specify, for each parameter, a discrete set of allowed values (e.g., values uniformly distributed within a range). The WG explores, with the *WL Navigator* component, several combinations of such parameters to find a combination able to reach the target state. The parameters represent the periodicity of the messages exchanged with the DS and other customizable factors, such as the *delays* to introduce in the processes for increasing the likelihood of sojourning in the target state for long enough. The WL Navigator makes use of a *Genetic Algorithm* (GA) to search for a suitable $w_c$. It starts from a random configuration, and then generates new candidate workloads by randomly mutating and combining candidate solutions

from a previous iteration, by replacing a parameter value of an existing solution with (i) another value from the set of allowed values, or (ii) a value of the same parameter taken from another workload. The quality (*fitness*) of $w_c$ is evaluated using a *fitness function*, which takes into account the "distance" between the tentative solution and the target states, and the "continuity" of periods spent in the target state. Based on $w_c$, the WG generates the actual workload $W$, by acting as a client of the SUT using a *WL Feeder* component.

### 4.2 The testing phase

After the workload search, in which a state-driven workload has been found, the system is actually tested using the selected workload. During a test (Fig. 2), events logs are still collected, and they are analyzed at run-time to trigger the Test Executor when the WG notices that the SUT has reached the target state. However, due to delays in the transmission of events and to the lack of clock synchronization, the test could be triggered in a global state that is different than the desired target state. In order to avoid incorrect experiments, we perform off-line synchronization after the test, analyze the execution report, and evaluate whether the test has been triggered in the correct global state. If the execution report points out that the test was triggered in an incorrect or undetermined state, the experiment is discarded and must be repeated. This "optimistic" approach is based on the observation that *if the system sojourns in the target state for long enough, it is likely that the test will triggered in a correct global state*, which is also assumed by other testing and fault injection tools for distributed systems such as the Loki [2]. Therefore, we can expect that tests will be correctly triggered most of times, and that only a few experiments will be discarded, as the WG seeks for a workload that maximizes the sojourn time in the target state. In any case, the off-line analysis assures that incorrect experiments are discarded and do not affect the evaluation of the system.

An important issue that we noticed in a preliminary implementation of our approach is that, even if the workload brings the SUT in the target state for a long enough time, it often happens that, during the same execution, the DS enters the target state only for short periods: in such cases, the test would be incorrectly triggered since the system leaves the target state during the execution of the test. In other terms, during an execution, there can be many state traversals shorter than the required $\tau$, and only a few traversals longer than $\tau$, where $\tau$ is the time required for the execution of the test (see Section 3). To mitigate this issue and to improve the likelihood of triggering the test in the desired global state, we adopt the following test triggering mechanism: we avoid (incorrect) triggering when a target state is traversed only for a short time, by raising the trigger only when *a triggering-delay $\theta$ has been elapsed since the system entered in the target state*. Figure 3 shows an example of the whole process, based on a hypothetical system model with two places and two transitions. The test trigger specification consists of the following conditions: (i) the place $p_2$ should have at least one token, and (ii) the first condition should hold for at least a delay $\theta^*$. During the testing phase, the WG collects events and updates its internal

representation of the global state. When an event happens, a message is sent to the WG, which updates the system state and checks if the target state (e.g., the marking $\langle p_1 = 0, p_2 = 1\rangle$) has been reached. If so, the $\theta$-delay starts, and the test is triggered only after $\theta$ is elapsed. Therefore, traversals of the target states shorter than $\theta^*$ will be filtered out. The delay $\theta$ is selected by the WG as follows, by maximizing the probability of correct test execution. A test is correct (i.e., triggered in the correct global state) if, after the SUT has reached the target state and remains in that state for a period $\theta$, it does not change state for an additional period $\tau$ to allow the Test Executor to perform the test. We estimate the *probability of test success* $pts_{\mathcal{S}_{G,\tau}}(\theta)$ by:

$$pts_{\mathcal{S}_{G,\tau}}(\theta) = \Pr\left(d_k \geq \theta + \tau \mid d_k \geq \theta \wedge s_k \in S_G\right) \cdot \\ \cdot \Pr\left(\exists e_k = (s_k, d_k) \in r_w : d_k \geq \theta \wedge s_k \in S_G\right) \quad (1)$$

where the first factor of the product represents the probability to stay in the target state $s_k$ for $\theta + \tau$ given that the triggering delay has been elapsed, and the second factor represents the probability that the workload will bring the system in target state for a long enough period at least once during the experiment. These probabilities can be empirically estimated from the execution reports collected during the workload search. The value of the triggering-delay $\theta^*$ is selected by the WG by maximizing the *pts*:

$$\theta^* = \arg \max_{\theta \in [0;\theta_{\max}]} pts_{\mathcal{S}_{G,\tau}}(\theta) \quad (2)$$

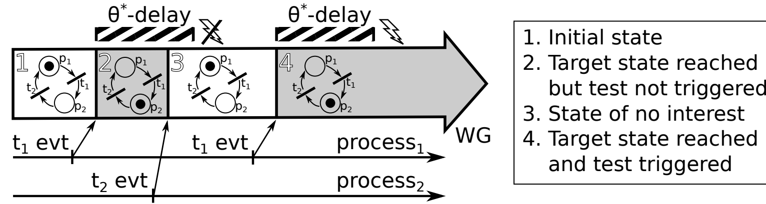


**Fig. 3.** Test triggering based on event logs and on a triggering delay.

## 5 Case study

We implemented and evaluated our approach within the Flight Data Processing System (FDPS) described in [14], and here summarized. FDPS is a distributed software developed in C++ which uses CARDAMOM, a fault-tolerant CORBA-compliant middleware. It is a part of an Air Traffic Control (ATC) system, in charge of managing and keeping up-to-date Flight Data Plans (FDPs).

The architecture of FDPS (Fig. 4) is composed by a Façade component, which acts as the frontend of the system and is replicated by the CARDAMOM Fault-Tolerance (FT) Service, and by a set of three Processing Servers (PSs),

managed by the Load-Balancing (LB) Service. Service requests are delivered to the Façade by the middleware: the Façade forwards requests to a specific PS according to a round robin scheduler; once the requests are completed, they are sent back to the Façade, which disseminates the updated FDP through a Data Distribution Service (DDS) and replies to the clients.

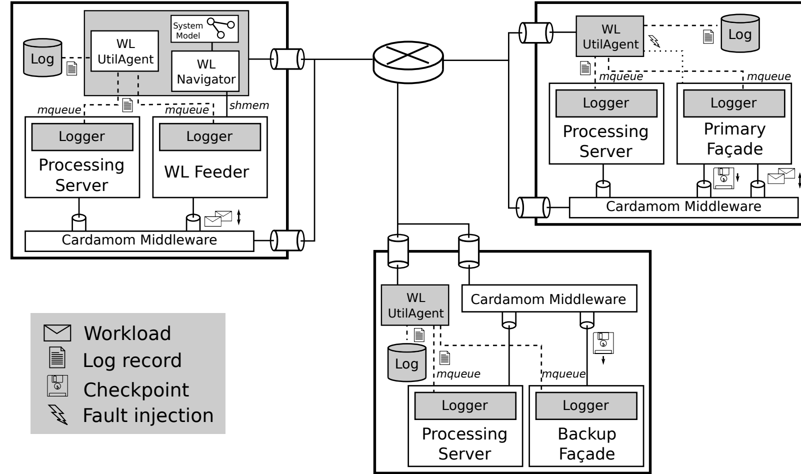


**Fig. 4.** The FDPS case study.

Requests are associated to a specific flight track, which is identified by an *FDP-ID* number: for each FDP-ID, the Façade dispatches no more than one request at time towards the PSs, and enqueues other requests for the same FDP-ID until the request under processing has been processed. The state of requests for each FDP is stored in an FDP Table of the Façade. Because the PSs are managed with a mono-threaded policy, the middleware in turn enqueues the requests forwarded to a PS if that PS is busy. The FT Service performs a *warm replication* of the Façade process: the FDP Table is checkpointed at each update and transmitted to backup replicas, which are activated in the case of failure of the primary replica. In our experimental setup, the application is installed on a LAN of RHEL Linux PCs connected through a 100Mbps Ethernet network; the FDPS deploys 3 Processing Servers, one active Façade and one backup Façade replica. The hosts adopted for the experiments were configured by disabling services that could interfere with the FDPS and the WG. In particular, we had to disable the NTP synchronization service, which modifies the system clock and can affect our synchronization algorithm [14].

In a previous study [3], we adopted fault injection to assess the fault tolerance of the warm replication mechanism implemented in the FDPS based on the CARDAMOM FT Service. The warm replication mechanism should copy the state of the FDP Table to a backup replica, and its effectiveness can be affected by the amount of data that has to be copied to the backup replica (i.e., the number of requests enqueued by the Façade) and by requests sent to PSs

(both under processing and enqueued by the middleware). In this case study, we perform fault injection experiments in different states of the FDPS, by taking into account the number of requests enqueued by the Façade and sent to PSs. We include in the system model of the FDPS (and thus in the definition of the global state) the number of requests enqueued at the Façade and at each PS. The system model, described in [28], was not included here due to space constraints.

In this experiment the workload configuration $w_c$ has a pair of parameters for each FDP-ID $i$, namely $T_{m_i}$ and $D_i$: the first one specifies the period between two consecutive requests sent by the client for the $i$-th FDP-ID; the second one represents a delay that is introduced in the PSs during the processing of requests for the $i$-th FDP-ID. The WL Feeder generates a *stream of requests* for each FDP-ID according to $w_c$. These parameters are communicated by the Workload Navigator to the Workload Feeder through a UNIX shared memory, whereas the Feeder transmits the delays $D_i$ to the Processing Servers in the request messages. Fig. 4 also includes the implementation of our state-driven testing approach in the FDPS (the shaded boxes in the figure). The *Logger*s are small libraries linked to FDPS processes; the CORBA objects were instrumented to collect application events by invoking Loggers, which in turn send event logs to the *WL UtilAgent*s using UNIX message queues. We log events that represent transitions in the system model. In particular, we log the invocation of CORBA methods (by invoking the Logger at the beginning of the CORBA method), and the accesses to request queues (e.g., by invoking the Logger when a private method for enqueuing requests is called). Event logs are processed by the WG (both during the search and the testing phases), which are translated into a sequence of global states: for instance, when a client request for the FDP #1 is received, a new global state is added in the sequence of global states, with $A1 = 1$ in the marking of the PN (see [28]). As an alternative to instrumenting the SUT, the events required by the system model can be obtained by system logs, if available. The *WL UtilAgent*s are processes that perform all the tasks required by the Workload Generator, such as log collection and off-line log analysis, and by the Test Executor, such as triggering a test. For instance, in our experiments we used the WL UtilAgent to inject faults in the Façade. We adopted the *process crash* as fault model, which is often adopted to evaluate the fault tolerance of distributed systems [2,7]. In our setup, the Test Executor is a process that forces a process crash, by killing the Façade process using UNIX signals. It is important to note that our approach can be adopted for injecting arbitrary fault models, depending on the type of system and on evaluation goals.

In the search phase, we configured the fitness function ([14], eqq. 4, 5) with parameters $\alpha = 10.4$ and $\varepsilon = 24$. A workload configuration $w_c$ represents an individual for the genetic algorithm, with $2 \cdot \#FDPs$ chromosomes (i.e., the parameters $T_{m_i}$ and $D_i$). At each iteration, the GA generates a new population of individuals (where each population consists of 8 individuals) from an old population, by repeatedly applying the following two rules:

- two individuals are randomly selected, with a probability based on their fitness; with probability $c = 90\%$ (*crossover rate*), the two individuals are

split in two parts (at a random point of the vector) and mixed (*crossover*), thus obtaining a new pair of individuals;
- with probability $m = 35\%$ (*mutation rate*), each parameter of the newly generated individuals are replaced (*mutation*), by randomly selecting a new value according to a normal distribution centered around the old value.

## 6 Experimental evaluation

We conducted a set of fault injection experiments on the FDPS, in order to evaluate the feasibility and effectiveness of the approach. In these experiments, we evaluate the ability and the speed of the WG to bring the system into a given target state, and the ability to correctly trigger fault injection while the system is in the target state. The target states are defined using a set of *constraints*, that is, conditions that the global state needs to satisfy: in the case of a system model based on Petri nets, the target states are represented by a set of conditions on the marking of the Petri net. If several global states satisfy the constraints, they are considered equally useful from the perspective of testing the DS. The Workload Generator is adopted for bringing the system in three different targets states, where each experiment introduces an additional constraint to the constraints of the previous experiment. Introducing additional constraints makes the search for a workload increasingly difficult, since each constraint reduces the set of target states. The experiments are defined as follows:

**Experiment #1**: The workload should bring the distributed system into a global state in which two out of three PSs are busy, and one out of three PS is idle. This condition is expressed by a constraint stating that the sum of tokens in the places $WRK_i$ (where $WRK_i = 1$ if the $i$-th PS is busy, and 0 otherwise [28]) should be exactly 2:

$$\sum_{i=1}^{3} WRK_i = 2 \tag{3}$$

**Experiment #2**: In addition to the previous constraint (Equation 3), the workload should bring the system into a global state such that the Façade should have enqueued at least 6 and at most 30 requests in its FDP Table (Equation 4). Both constraints should hold at the same time in order to reach the target state. The second constraint states that the sum of tokens in the places $A_j$, representing the number of requests in the FDP queue $j$ (with six FDP queues in total) [28], should be between 6 and 30 tokens:

$$6 \leq \sum_{j=1}^{6} A_j \leq 30 \tag{4}$$

**Experiment #3**: The set of target states is further restricted, by (i) including the constraint of experiment #1 (Equation 3), (ii) replacing the constraint of Equation 4 with the more restrictive condition of Equation 5, and (iii)

adding the condition that there should be at least one request enqueued by the PSs (Equation 6). All three constraints should hold at the same time in order to reach the target state. Equation 5 states that each FDP queue $j$ should have between at least 1 and at most 5 enqueued requests (instead, Equation 4 disregards how enqueued requests are distributed across FDP queues); in Equation 6, tokens in the place $BF$ represent requests enqueued by the middleware for the PSs, which should be more than zero [28]:

$$1 \leq A_j \leq 5 \ , \qquad j = 1, 2, \ldots, 6; \tag{5}$$

$$BF > 0 \tag{6}$$

It is important to note that, even though the first and the third constraints (Equations 3 and 6) appear to be contradictory, it is in fact possible to satisfy them at the same time. These constraints state that there should be an idle Processing Server, while the other two PSs should be busy and have requests enqueued for them, i.e., the enqueued requests should not be forwarded to the idle PS. This condition is actually possible since the request scheduler selects the PS for an incoming request on a round-robin basis, regardless of whether the selected PS is busy and whether there are idle PSs. Therefore, this condition is hard-to-reach, but possible.

We imposed a minimum sojourn time in the target state of $\tau = 0.3s$, which is large enough to allow our Test Executor module to be triggered and to kill the Façade process. We fixed the number of FDP-IDs to six, and set the domains for the parameters of the workload configuration ranging from 500ms to 5s, with a step of 500ms. Finally, we set $|\mathcal{R}_W| = 3$.

In Fig. 5, we depict the sojourn time in the target state attained by generated workloads. At each iteration of the genetic algorithm, a generation (i.e., set of solutions) is obtained by mutating and combining solutions from the best solutions of the previous generation (on the basis of the fitness function). We evaluated, for each generation, the sojourn time attained by the best solution of each generation. In every experiment, the WG was able to find a workload able to bring the system into the target state for an uninterrupted time period of at least $1.5s$; the convergence to a "good" solution was very quick in the case of experiments #1 and #2, and in the case of experiment #3, which imposed more restrictive constraints to the target state, the algorithm converged after 14 iterations, which were executed in about 3 hours.

For each experiment, we selected the workload with the highest uninterrupted sojourn times across all generations of the search, and then we used that workload for fault injection experiments.

The table 1 shows the probabilities of correct test execution and compares them with the estimations obtained by $pts_{\mathcal{S}_G,\tau}(\theta^*)$ (eq. 1). The probabilities of correct test executions has been obtained by performing 100 fault injection experiments on the system for each target state, and by evaluating whether faults where injected in the correct global state: most of fault injection experiments

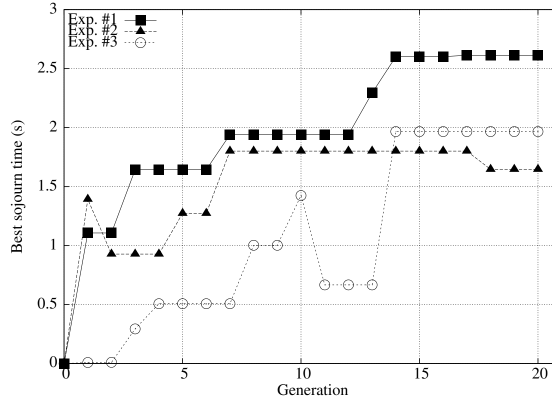


**Fig. 5.** Sojourn time of the best solution at each generation of the genetic algorithm.

were correctly performed, with a probability of correctly reproducing the experiment of 60% in the worst case. In every case, the estimated probability of test success was close to the probability actually experienced during experiments, with a difference less than 10%. Since the probability of test success is high, it is likely that the test is performed in the correct state on the first try, or after a small number of repetitions.

**Table 1.** Probability of injecting a fault in the correct global state.

| | **Exp. #1** | **Exp. #2** | **Exp. #3** |
|---|---|---|---|
| *Experimental test success probability* | 82.6% | 82.9% | 57.1% |
| *Predicted test success probability* | 92.2% | 75.0% | 60.0% |

We analyzed the overhead of our WG approach on fault injection experiments, by evaluating the performance loss due to our instrumentation. The only instrumentation we introduced was the logging of events in the FDPS, and the collection of these events in order to trigger the injection of faults. Fig. 6 shows the average response time of the FDPS over 20 executions, at different rates of input requests, when logging and collection are disabled and enabled, respectively. The increase of the request completion time is 4% in the worst case, and is less significant at higher rates of input requests. Therefore, the performance overhead incurred during execution with instrumentation can be considered negligible, meaning that the program behavior remains realistic during an experiment.

## 7 Conclusion

The global state is a major concern in the verification of a distributed system. State-driven testing of distributed systems proves to be challenging due to system complexity, the use of OTS components, the clock drift and the non-determinism

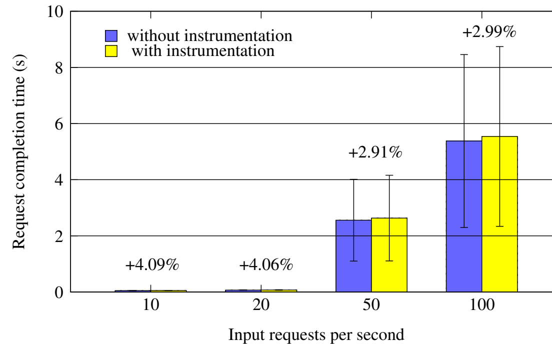


**Fig. 6.** Overhead of event log collection and processing.

of distributed systems. We proposed an approach for state-driven testing of complex distributed systems, that automates the search for a state-driven workload, and perform tests in a desired global state with probabilistic guarantees.

**Acknowledgments.** This work has been supported by the project "Embedded Systems in Critical Domains" (CUP B25B09000100007), by the TENACE PRIN Project (n. 20103P34XC) funded by the Italian Ministry of Education, University and Research, and by the Finmeccanica industrial group in the context of the project "Iniziativa Software CINI-Finmeccanica".

## References

1. Lee, I., Iyer, R.: Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System. In: Proc. 23th Symp. on Fault-Tolerant Computing. (1993) 20–29
2. Chandra, R., Lefever, R., Joshi, K., Cukier, M., Sanders, W.: A Global-State-Triggered Fault Injector for Distributed System Evaluation. IEEE Trans. Parallel and Distributed Sys. **15**(7) (2004) 593–605
3. Natella, R., Cotroneo, D.: Emulation of transient software faults for dependability assessment: A case study. In: Proc. Eur. Dependable Comp. Conf. (2010) 23–32
4. Lefever, R., Cukier, M., Sanders, W.: An experimental evaluation of correlated network partitions in the Coda distributed file system. In: Proc. Intl. Symp. Reliable Distributed Systems. (2003) 273–282
5. Vieira, M., Madeira, H.: A dependability benchmark for OLTP application environments. In: Proc. 29th Intl. Conf. on Very Large Data Bases. (2003) 742–753
6. Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J., Laprie, J., Martins, E., Powell, D.: Fault injection for dependability validation: A methodology and some applications. IEEE Trans. Software Eng. **16**(2) (1990) 166–182
7. Meling, H., Montresor, A., Helvik, B., Babaoglu, O.: Jgroup/ARM: a distributed object group platform with autonomous replication management. Soft.: Pract. Exp. **38**(9) (2008) 885–923
8. Tsai, T., Hsueh, M., Zhao, H., Kalbarczyk, Z., Iyer, R.: Stress-Based and Path-Based Fault Injection. IEEE Trans. Computers **48**(11) (1999) 1183–1201
9. Kiskis, D., Shin, K.: SWSL: A synthetic workload specification language for real-time systems. IEEE Trans. Soft. Eng. **20**(10) (1994) 798–811

10. Duraes, J., Madeira, H.: Generic faultloads based on software faults for dependability benchmarking. In: Proc. Intl. Conf. Dependable Systems and Networks. (2004) 285–294
11. Kalakech, A., Kanoun, K., Crouzet, Y., Arlat, J.: Benchmarking the Dependability of Windows NT4, 2000 and XP. In: Proc. Intl. Conf. Dependable Systems and Networks. (2004) 681–686
12. Zhang, F., Cheung, T.y.: Optimal transfer trees and distinguishing trees for testing observable nondeterministic finite-state machines. IEEE Trans. Soft. Eng. **29**(1) (2003) 1–14
13. Nachmanson, L., Veanes, M., Schulte, W., Tillmann, N., Grieskamp, W.: Optimal strategies for testing nondeterministic systems. ACM Soft. Eng. Notes **29**(4) (2004) 55–64
14. Natella, R., Scippacercola, F.: Issues and Ongoing Work on State-Driven Workload Generation for Distributed Systems. In: Dependable Comp. Springer (2013) 96–110
15. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability (2012)
16. Bourhfir, C., Dssouli, R., Aboulhamid, E., Rico, N.: Automatic executable test case generation for extended finite state machine protocols. Test. Comm. Sys. (1997) 75–90
17. Kerbrat, A., Jéron, T., Groz, R.: Automated test generation from SDL specifications. In: Proc. 9th SDL Forum. (1999) 135–152
18. Kiskis, D.L., Shin, K.G.: A synthetic workload for a distributed real-time system. Real-Time Systems **11**(1) (1996) 5–18
19. Weyuker, E.J., Vokolos, F.I.: Experience with performance testing of software systems: issues, an approach, and case study. IEEE Trans. Soft. Eng. **26**(12) (2000) 1147–1156
20. Arlat, J., Aguera, M., Crouzet, Y., Fabre, J., Martins, E., Powell, D.: Experimental evaluation of the fault tolerance of an atomic multicast system. IEEE Trans. Reliab. **39**(4) (1990) 455–467
21. Basile, C., Wang, L., Kalbarczyk, Z., Iyer, R.: Group communication protocols under errors. In: Proc. Intl. Symp. Reliable Distributed Systems. (2003) 35–44
22. Dawson, S., Jahanian, F., Mitton, T., Tung, T.: Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In: Proc. Fault Tolerant Computing Symp. (1996) 404–414
23. Hoarau, W., Tixeuil, S.: A language-driven tool for fault injection in distributed systems. In: Wksp. Grid Comp. (2005) 194–201
24. Helvik, B., Meling, H., Montresor, A.: An approach to experimentally obtain service dependability characteristics of the Jgroup/ARM system. Proc. Eur. Dependable Comp. Conf. (2005) 179–198
25. Joshi, K., Cukier, M., Sanders, W.: Experimental evaluation of the unavailability induced by a group membership protocol. Proc. Eur. Dependable Comp. Conf. (2002) 644–648
26. Poirier, B., Roy, R., Dagenais, M.: Accurate offline synchronization of distributed traces using kernel-level events. ACM SIGOPS Operating Systems Review **44**(3) (2010) 75–87
27. Scippacercola, F.: State-Driven Workload Generation in Distributed Systems. Master's thesis (2012)
28. Cotroneo, D., Natella, R., Russo, S., Scippacercola, F.: State-driven testing of distributed systems: Appendix. Technical report (2013) Available at: `http://www.mobilab.unina.it/techreports.html`.