Issues and Ongoing Work on State-Driven Workload Generation for Distributed Systems

Roberto Natella[1] and Fabio Scippacercola[2]

[1] Università degli Studi di Napoli Federico II
roberto.natella@unina.it
[2] Consorzio Interuniversitario Nazionale per l'Informatica
fabio.scippacercola@consorzio-cini.it

Abstract. The dependability of a complex distributed system needs to be assured against the several conditions, namely *states*, in which it can operate. Generating a workload able to cover a desired target state of a distributed system is still a difficult task, since the relationship between the workload and states is nontrivial due to system complexity and non-deterministic factors. This work discusses our ongoing work on a state-driven workload generation approach for distributed systems, based on an evolutionary algorithm, and its preliminary implementation for testing a fault-tolerant distributed system for flight data processing.

Keywords: Distributed Systems; State-based Testing; Fault Tolerance; Fault Injection; Workload; Genetic Algorithms; Off-line synchronization

1 Introduction

In order to assess and to improve the dependability of a complex distributed system, verification techniques have to inspect those operating conditions, namely *states*, that can expose the system to failures.

The relationship between the application states and dependable behavior was shown in several past studies [1,2,3]. This is particularly important in the case of *fault injection*, as revealed by several studies on the assessment through fault injection of distributed filesystems [4,5], DBMSs [6], and multicast and group membership protocols [7,8,5]. These studies emphasized that the success of recovery is influenced by the state of the distributed system. For instance, if we consider a DBMS that has to guarantee the ACID properties to distributed transactions, its recovery mechanisms (e.g., the rollback to a previous state) can be affected by several factors, such as the presence of several transactions that access to the same resources or that are nested. It is thus evident how a *state-driven workload*, i.e., a workload that brings the system in target states during the analysis, is important to assure the significance and the efficiency of experiments, by *covering the states where the system has to be tested*.

It is well-known that generating a state-driven workload for distributed systems is a difficult and time-consuming task, since the relationship between the workload and states is nontrivial, due to system complexity and non-deterministic

factors, such as concurrency and network delays. Past studies proposed the generation of synthetic randomly-generated workloads [9,10], or relied on realistic workloads derived from performance benchmarks [6,11,12]. Nevertheless, these approaches are not meant to cover hard-to-reach. Other approaches generate a workload from stochastic or non-deterministic models of the system, but do not scale well for complex systems [13,14]. Therefore, the automatic generation of state-driven workloads is a relevant but still open issue in distributed systems.

This paper discusses our ongoing work on the automatic generation of state-driven workloads for distributed systems. We propose the use of an agent, which iteratively explores the space of workloads using an evolutionary algorithm. At each iteration, the agent modifies the workload and evaluates its goodness, until the system converges to the target states. To this aim, the approach allows to specify the desired mean probability of reaching the target state for a specified amount of time, thus enabling the injection of faults in the target state. The approach is fully automated and does not rely on a detailed characterization of the relationship between workloads and states, and can therefore be adopted for testing the *actual implementation* of complex distributed systems. We also discuss a preliminary implementation of the approach for testing a fault-tolerant distributed system for flight data processing.

The paper is organized as follows. Section 2 discusses previous studies on state-based and fault injection testing of distributed systems. Section 3 provides basic concepts and assumptions, and Section 4 describes the proposed approach. Section 5 describes how the approach has been implemented in a real-world distributed system, and provides preliminary results. Section 6 concludes the paper and describes future developments of the work.

2 Related work

While the problem of testing stateful non-distributed systems has been studied in depth [15], the state-based testing of distributed system poses additional and still unsolved challenges, in particular in testing the *actual implementation* of a distributed system. Studies on the verification of distributed systems can be classified into two classes: the analytical-simulation studies and the experimental ones.

Analytical and simulation studies are based exclusively on *analytical* or *behavioral* descriptions of the system, such as Finite State Machines (FSM), Petri Nets (PN), and Computational Tree Logic (CTL). They assess properties or conditions of the system through mathematical proofs, simulations or model checking methods on models [16]. These approaches require abstract models of the system, which have to be hand-written by the tester, or extracted from the system [17]. They are suitable for the verification of the high-level design of the system (e.g., testing protocols or distributed algorithms), but need to be complemented with experimental approaches in order to test low-level design and implementation aspects of the system.

Experimental studies, in which our work is included, exercise and asses the actual implementation of a system, by allowing to analyze a system during its execution. 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 [7]. In these studies, the problem of the state and of workload generation has been approached in several ways.

In some studies, a model of the system is adopted for automatically generating test cases for the system. For instance, conformance testing approaches generate test cases aimed at covering the states of the model and at assuring that the system evolves as described in the model. These approaches are based on a detailed model of the target system, which describes the expected behavior[18,19,20]. Since in most cases distributed systems are non-deterministic (i.e., the system may evolve in more than one way given the same inputs, due to random factors), the model is also non-deterministic. Some approaches, such as those described in [13,14], generate sequences of inputs able to drive the system state in spite of random factors, but their application in complex systems is limited by scalability issues due to the space explosion problem, and by restrictive assumptions they implicitly make about the behavior of the system (for instance, they only consider "stable" states, in which the system waits for inputs or events [18]).

Other studies, including ones on fault injection, do not rely on a system model to generate a workload, but they assess its performance or dependability by adopting a workload representative of the real system workload that will be experienced during operation [6,11,12], in a similar way to performance benchmarks of non-distributed systems [21]. In other cases, synthetic workloads are randomly generated, in which the tester provides a probability distribution over the input space of the system [9,22,23], or provides a high-level description of the synthetic workload, e.g., using the *Synthetic Workload Specification Language* [24]. In such studies, the system states that are tested are only those ones exercised by the considered workload, and they do not consider the problem of tuning the workload in order to bring the system in "hard-to-reach" states. In particular, many fault injection studies randomly inject faults during an experiment, repeating this process several times and performing a very high number of experiments [25,26,8], which can be ineffective at uncovering vulnerable states of the system. More sophisticated fault injection approaches trigger the injection when a specific state of the system occurs [27,28,5]. For instance, Loki [5] considers the global state of a distributed system for triggering fault injection: in order 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. However, 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.

3 Basic concepts and assumptions

According to the traditional definition [29], a Distributed System (DS) is composed by *processes*, which execute concurrently on a set of nodes, and by a *network*, which is the only medium through which the processes interact. Each process has its own clock and encapsulates some local resources. Local resources cannot be read or updated by any other process without an explicit request. Therefore, processes cannot share memory and the interactions among them only occur through messages exchanged on the network.

In order to design an approach for generating state-driven workloads, we make some practical assumptions about the architecture of a distributed system. We consider distributed systems in which a set of *services* is exported by a *frontend* process, which masks 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. This view of DSs applies to several systems, including orchestrated web services and three-tier web applications [29].

More formally, the frontend exports a set of services $\mathcal{U} = \{u_1, u_2, \ldots, u_n\}$ by an *interface*: each service can be invoked by the clients and it triggers different functionalities and actions in the DS depending on the actual values of the parameters sent in its requests. Without loss of generality, we assume that the tester defines the sets M_{u_i} before the assessment of the target system, where $m_{i,j} \in M_{u_i}$ is the j-th combination of parameters for the service u_i that a client can invoke on the frontend. A *service request* for the service u_i is a message produced by the client with parameters $m_{i,j}$ at time $t \in \mathcal{T} = \{0, \ldots, t_{\max}\}$ of the experiment, and can be represented by a pair $r \in M_{u_i} \times \mathcal{T}$. A *workload* is a set of service requests generated during an execution, and it is a subset W of the set W^* representing the space of all possible requests that can be submitted to the system: $W \subseteq W^* = \bigcup_{u_i \in \mathcal{U}} \{(m_{i,j}, t), m_{i,j} \in M_{u_i}, t \in \mathcal{T}\}$. In other words, a workload is an element of the *powerset* (the set of all subsets) of W^*, that is, $W \in \mathcal{P}(W^*)$.

The aim of the Workload Generator (WG) is to select a $\overline{W} \in \mathcal{P}(W^*)$ such that the DS reaches a *target state* (or any state from a *set of target states*), specified by the tester, during the execution of $\overline{W}$. The state of an individual process in the DS is referred to as *local state* of the process, whereas the *global state* of the DS, denoted with $s \in \mathcal{S}$, is the union of all the local states. The state of the process and of the DS is determined by the tester according to some high-level specification of the system, which takes into account the state of local resources as well as the state of computations performed by each process. An example of local state of a process could be DOWN if the process is failed, or UP otherwise; or INITIALIZING and WAITING FOR ACK to distinguish between different states of a computation. The global state of the system is specified by the tester through a *system model*. For instance, if we want test the correctness of a deadlock detection mechanism in a distributed DBMS, the system model and the global state would reflect the contents of the lock table and the distributed

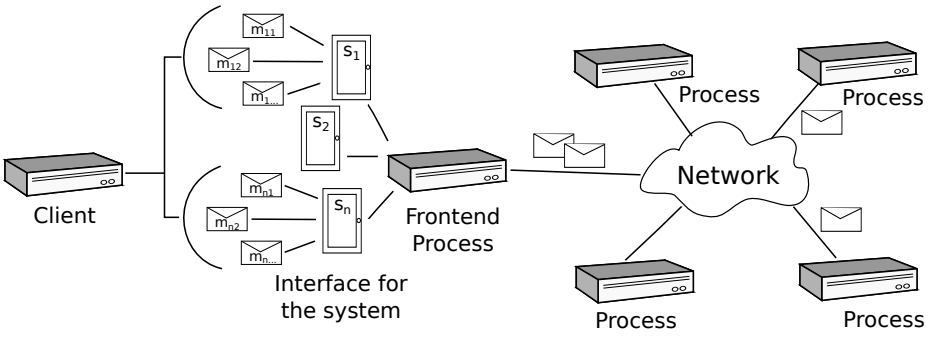

Fig. 1. The distributed system architecture considered in this work.

transactions being performed. The system model could be represented using any formalism, such as Finite State Machines and Petri Nets.

A distributed system is intrinsically concurrent. The rate at which each process executes and the timing of the messages exchanged on the network are unknown. Moreover, the exchange of messages through the network is affected by several factors, including the delays for accessing to the network and for transmitting the contents of messages, and the delays introduced by the OS and by a middleware layer at both the ends of a communication. It follows that it is often impractical to predict the evolution of a DS, since it is difficult to precisely model all the factors that affect the system, especially when the system in the very complex and includes third-party and off-the-shelf hardware and software components. Moreover, the same sequence of client requests can produce different evolutions of the system due the randomness of these factors.

When an experiment is executed, the workload W causes the system to traverse one or more states, and to sojourn in each of them for a finite time. Let the *execution report* be the sequence $\{e_n\}_{n\in\mathbb{N}}$, where $e_n = (s_n, d_n)$ represents the state s_n traversed by the system during its n-th evolution of an execution, with a sojourn time d_n. Let $\mathcal{S}_G \subset \mathcal{S}$ be the subset of target states in which the tester aims to bring the distributed system: the minimum sojourn time τ is the time required by the tester to evaluate a property of the target system in $\mathcal{S}_G$, and represents a constraint for the WG. The *target hit ratio*, $p_{\mathcal{S}_G,\tau}(\mathcal{R}_W)$, applied on the set of execution reports $\mathcal{R}_W = \{r_1, \ldots, r_N\}$ obtained from one or more executions under workload W, estimates the probability that the workload W brings the system in the target state for a sojourn time greater than τ during an execution:

$$\begin{aligned} p_{\mathcal{S}_G,\tau}(\mathcal{R}_W) &= \mathcal{P}\{\text{The DS reaches } \mathcal{S}_G \text{ for more than } \tau \text{ at least one time when executing } W\} \\ &= \frac{|\{r_i \in \mathcal{R}_W : \exists (s_k, d_k) \in r_i : (s_k \in \mathcal{S}_G) \wedge (d_k > \tau)\}|}{|\mathcal{R}_W|} \end{aligned} \tag{1}$$

where the $|\cdot|$ operator represents the cardinality of a set. For instance, if the fault tolerance of the DS is being evaluated through fault injection, the DS has to sojourn in the target state long enough to allow a fault injection tool to detect the state and to inject a fault before the DS leaves the target state. In this scenario, $p_{\mathcal{S}_G,\tau}$ would represent the likelihood of a correct fault injection experiment, i.e., the fault is injected while the system is in the desired state.

The problem of generating a state-driven workload consists in searching for a workload $\overline{W}$ such that

$$p_{\mathcal{S}_G,\tau}(\mathcal{R}_W) > p_d \ , \tag{2}$$

that is, the likelihood to spend a period τ in the target state is high enough (i.e., greater that p_d) to allow an accurate and reproducible test. The search is conducted by the tester before performing the desired test. Eq. 2 provides a stop criterion for the search. Then, the tester supplies again $\overline{W}$ to the system, and performs the test when the system reaches a target state (e.g., it performs the actual fault injection experiment).

4 Proposed approach

The proposed approach is based on a Workload Generator (WG) agent that interacts with the Distributed System Under Test (DS-UT) in a closed-loop configuration, as shown in Figure 2. The WG exercises the DS with a workload, analyzes its behavior, and modifies the workload until a specified target state is reached.

The WG follows a *system model* of the DS, which is adopted by the tester to specify the states of the system, and enables the WG to understand whether a target state has been reached (i.e., the system model is adopted for computing the execution reports from the raw events occurred and collected during the execution).

The WG works iteratively, by alternating at each iteration an *off-line* and an *on-line* phase. In the on-line phase, the WG executes the DS several times. At each execution, it first brings the DS in its initial state s_0 through a *reset* operation, it feeds the DS with a workload W, and observes its evolution for a fixed time period. Then, after the DS has stopped, there is an off-line phase in which the WG analyzes the behavior of the system through *logs* collected during execution, and evaluates whether the target state has been reached. If this was not the case, a new workload is computed and used in the next iteration. The distinction between the off-line and on-line phases allows to reduce the intrusiveness of the WG on the DS under test, since only minimal information is collected during the execution of the system, and most of the processing for analyzing the system evolution and computing the workload occurs in the off-line phase.

We divide the discussion of the proposed approach in three parts: *modeling* the system states, *monitoring* the execution of the DS, and *driving* the DS by tuning the workload.

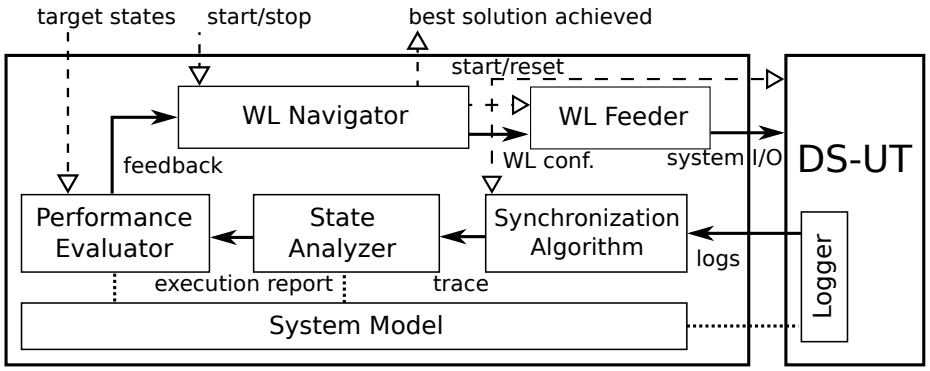

Fig. 2. The architecture of the proposed Workload Generator. Dotted lines represent a relation of dependence, while arrows represent dynamic interactions: striped arrows represent a control flow; continuous ones denote a data flow.

Modeling

Our approach is based on a *system model* of the DS under test, which allows to compute the execution report $\{(s_n, d_n)\}_{n\in\mathbb{N}}$, and to evaluate how close the workload brings the system to the target states.

Since the complexity of the system under test is typically very high, we assume that the system model provides an abstract and simplified view of the system. In particular, we do not require the system model to characterize the *timing of events* in the system, but only the *relationship* between events and states. As discussed in Section 3, timings in complex distributed systems, including communication and computation delays, can be unfeasible to characterize even in a probabilistic way, since they are tightly depending on each other, and involve third-party and off-the-shelf components, whose internals are unknown. Moreover, the timing of events also depends on the state of the system and on its workload (e.g., communication delays depend on the number of processes accessing the network at a given time): since the workload is iteratively modified by the WG to drive the DS in the target state, the characterization of delays would not hold when the workload is changed by the WG.

Petri Nets (PN) are the formalism that we adopt for modeling the DS under test, as they are a popular formalism that fits well for modeling concurrent systems. In the system model, transitions are triggered by local events occurring at the processes of the DS. Since the timings of events are not modeled, transitions are not timed, but only express the relationship between events and the state of the DS. The state is represented by the marking of the PN. In our approach, the system model is used in the off-line phase (*after* exercising the DS using a workload) to obtain, from raw event logs of an execution, the sequence of states that the system has followed during the execution.

Monitoring

The monitoring is realized by the Logger and the Synchronization Algorithm (Fig. 2). The process for generating an execution report is summarizing in Fig. 3. Each process of the DS logs the local events, timestamping the records with its own clock. When the experiment is over, the logs of local events are collected, and an offline synchronization algorithm, which is described later, performs the temporal sorting. Then, the State Analyzer obtains a sequence of markings of the system model, by mapping events in the logs to transitions in the model, and generates the execution report.

Regarding the synchronization, in order to obtain the evolution of the system from the analysis of events, we adopt an *off-line synchronization algorithm* to align the events of an execution on a single global timeline [5,30]. Off-line synchronization has been preferred over on-line synchronization approaches [29], such as NTP, since on-line synchronization protocols exchange packets during the execution of the system and can thus interfere with its evolution. Off-line synchronization is performed after execution, by correcting the timestamps of events recorded during the execution. The correction is performed using an estimate of the drift rate of the clocks, which is obtained by analyzing the round-trip time of a set of messages exchanged before and after the execution. Since the clock drift rate can only be estimated, the exact timing of an event is unknown; instead, the off-line synchronization algorithm provides, for each occurred event, a *lower* and an *upper bound* for the event on the global timeline, which represent the *uncertainty interval* in which the event has occurred. When the uncertainty intervals of two events are not overlapped, their ordering and the evolution of the system can be determined. When overlaps occur, the state of the DS is unknown in the overlapped region of the global timeline. More details about off-line synchronization algorithms can be found in [5,30].

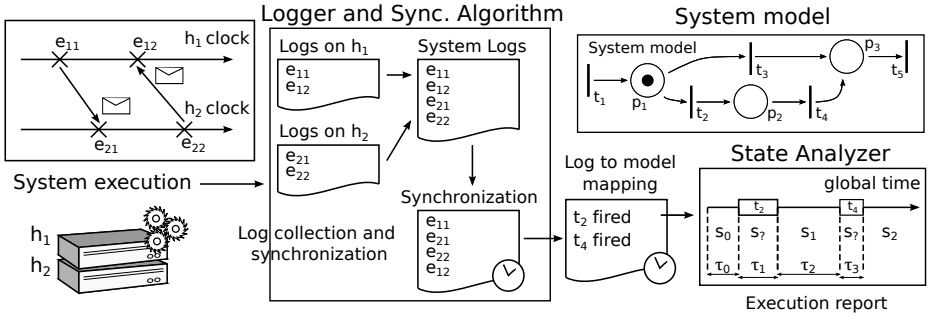

Fig. 3. The process for issuing execution reports. When the State Analyzer cannot determine the system state, due to the synchronization accuracy, it assigns the dummy state $\mathcal{S}_?$.

Driving

In our approach the WL Navigator and the WL Feeder drive the DS in the target states. The Feeder is the active part of the agent, which interacts with the DS generating the requests, i.e., the workload. The workload is synthesized by the Feeder on the basis of a *workload configuration* $w_c \in W_c$, which characterize the timing and the type of requests, either deterministically or statistically, through a set of parameters. The Navigator is the "smart" part of the agent, which searches for a proper workload configuration and analyzes the feedback from the system.

The distinction between Feeder and Navigator eases the search process, as the number of parameters in a workload configuration is a compact representation of a set of request $W \in \mathcal{P}(W^*)$ (actually, $|W_c| \ll |\mathcal{P}(W^*)|$), and frees the search algorithm from subtle details about individual interactions with the DS.

Herein, we consider a workload configuration $w_c \in W_c$ defined the following vector of parameters:

$$w_c = \left\langle T_{m_{1,1}}, T_{m_{1,2}}, \ldots, T_{m_{N,1}}, T_{m_{N,2}}, \ldots, D_{p_1}, \ldots, D_{p_M} \right\rangle \ . \tag{3}$$

These parameters are used by the Feeder to generate a set of requests $W \subseteq W^*$ to submit to the DS. In our case, the Feeder periodically invokes each service u_i using parameters $m_{i,j}$, with a period of $T_{m_{i,j}}$, $\forall m_{i,j}$. The Navigator explores several combinations of values for the $T_{m_{i,j}}$ parameters, in order to find a combination able to reach the target state. Moreover, we consider an additional set of parameters, D_{p_k}, which represent *delay factors* to introduce in one or more processes in the DS. Since the system may evolve very quickly, the target state could be reached only for short periods of time, leading to a low probability of hitting the target state for a sufficient time (Eq. 2). The introduction of small delays, by either slowing down a process (e.g., reducing its CPU quota by tuning its scheduling priority) or by forcing the process to sleep for short periods of time, increases the likelihood of sojourning in the target state for long enough.

The search for a proper workload proceeds through iterations. At each iteration, the Navigator changes the workload configuration on the basis of the feedback of the previous iterations, until a target state has been reached. To do so, two elements need to be defined, namely (i) a search algorithm to explore the space of workload configurations W_c, in order to find a workload $\overline{W}$ suitable for reaching the target state, and (ii) a criterion for assessing the "quality" of the current workload with respect to the target state.

We adopted in the Navigator a *genetic algorithm* (GA) [31]. A genetic algorithm evaluates a *population* of solutions (*individuals*) during the off-line phase of each iteration; then, a new population is generated from the previous one, by randomly *mutating* and *combining* previous individuals, on the basis of their quality (*fitness*).

Each individual of the population represents a workload configuration. An individual w_c is evaluated by executing the system under the workload W, generated by the Feeder using w_c, and by evaluating a *fitness function* on the execution traces, which is computed by the *Performance Evaluator* component (Fig. 2).

The definition of the fitness function is an important aspect of our approach, since it drives the Navigator in the search for the target state. The $p_{\mathcal{S}_G,\tau}(\mathcal{R}_W)$ (Eq. 2) cannot be used as fitness function, as it is not able to compare two different solutions that do not reach the target states (it would be 0 for both solutions). Instead, we propose a fitness function that evaluates the "distance" between the tentative solution and the target states, and the "continuity" of periods spent in the target state. Considering a set of execution reports obtained from one or more executions under workload W, $\mathcal{R}_W$, the fitness function is defined as

$$f_{\alpha,\varepsilon}(\mathcal{R}_W) = \frac{1}{|\mathcal{R}_W|} \sum_{r_w \in \mathcal{R}_W} \left(\sum_{(s,d) \in r_w} d^{\alpha} \cdot 10^{-\varepsilon \cdot \mathrm{dist}(s)} \right) \tag{4}$$

where the α rewards the continuity of the sojourn times (*permanence bonus*), while the ε penalizes the distance from the target states (*distance bonus*). When two solutions have a different distance, the closest solution is privileged (the *distance bonus* predominates); when two solutions have the same distance, the most continuous solution is privileged (the *permanence bonus* predominates). The times are normalized, i.e., $\sum_{(s,d) \in r_w} d = 1 \quad \forall r_w \in \mathcal{R}_W$ and $d \geq 0 \quad \forall (s,d) \in r_w$.

The function *dist* is a *distance measure* between any state and the target states, which is introduced to reward the workloads that are closer to the target states. We propose two different measures for the Petri net model we have adopted:

1. The minimum difference of tokens between the marking of the actual state and any target state: this measure is coarse and imprecise, however, is fast to calculate and it is easy to understand. Given a vector marking M, and let G be the set of target markings of a PN with m places, we have:

$$\mathrm{dist}(M, G) = \arg \min_{M' \in G} \left(\sum_{0 \leq i < m} |M_i - M'_i| \right) \tag{5}$$

2. The difference in the breadths on the reachability graph between the state and any target state, i.e., the minimum number of transitions (events) that have to happen to reach the closer target state. To compute this distance, we need to do a breadth-first search in the PN. Since we might not reach any target states, and because we do not need a precise value in the applications if the distance is greater than a fixed threshold, then we can limit the expansion of the PN at a depth R. Let BFS be an R-bounded breadth first visit, we have the follow:

$$\mathrm{dist}(M, G, R) = \arg \min_{M' \in G} \left(\mathrm{BFS}\left(M, M', R\right) \right) \tag{6}$$

   In particular, we could save in memory all the states that are within range R from all the target states, avoiding to search in a graph each time. Whenever

we would need to compute the distance of a node, we would just need to check if M is in memory. If there is, then we would know the distance, else $dist(M, G, K) > R$.

To select the α and ε parameters of the fitness function, the following heuristics can be adopted. Considering any two states $s_l, s_{l+1} \in \mathcal{S}$ such that $\text{dist}(s_{l+1}) = \text{dist}(s_l) + 1$, it is possible select the parameters α, ε, according to the following system of inequalities:

$$\begin{cases} f_{\alpha,\varepsilon}\left(\{r_w^1 = \{(s_l, \theta \cdot \tau)_1\}\}\right) \geq f_{\alpha,\varepsilon}\left(\{r_w^2 = \{(s_l, \tau)_1, \ldots, (s_l, \tau)_k\}\}\right) \\ f_{\alpha,\varepsilon}\left(\{r_w^3 = \{(s_l, \eta)_1\}\}\right) \geq f_{\alpha,\varepsilon}\left(\{r_w^4 = \{(s_{l+1}, 1)_1\}\}\right) \\ \theta > 1;\ 0 < \eta < 1 \end{cases} \tag{7}$$

The first inequality expresses that it is better to sojourn in a state s_l for a $\theta \cdot \tau$ time, rather than visiting the same state up to k times for the same τ time. In this way, the tester can control the amount of reward to provide through the permanence bonus. The second inequality expresses that it is better to remain at least a η time in a state s_l, rather than to stay full-time in a state s_{l+1}, one level farther from the target states; this inequality can be adopted to control the level bonus. Developing the system we obtain the following relations:

$$\begin{cases} \frac{(\theta\tau)^\alpha}{10^{\varepsilon l}} \geq \frac{k\tau^\alpha}{10^{\varepsilon l}} \Rightarrow \alpha \geq \frac{\log k}{\log \theta} \\ \frac{\eta^\alpha}{10^{\varepsilon l}} \geq \frac{1}{10^{\varepsilon(l+1)}} \Rightarrow \varepsilon \geq -\alpha \log \eta \end{cases} \tag{8}$$

For instance, if we prefer staying in a state for a 25% longer time instead of visiting ten times a state for the same duration, and if we want that the sojourning for the 1% of total time in a "close" state s_l is better than staying for the 100% of total time in a "far" state s_{l+1}, then we have: $\alpha \geq \frac{\log 10}{\log 1.25} \geq 10.318$, $\varepsilon \geq -\alpha \log 0.01 = 2\alpha \geq 20.636$.

5 Preliminary implementation

Herein, we present a preliminary implementation of our approach for testing a Flight Data Processing System (FDPS). 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 Flight Data Plans (FDPs). An FDP is a data structure containing information about a flight; the goal of FDPS is to keep FDPs up-to-date. For example, FDPS has to analyze the actual position of aircrafts, retrieved from radar tracks, and update flight routes consequently, in order to efficiently allocate the flight space and to avoid flight collisions.

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

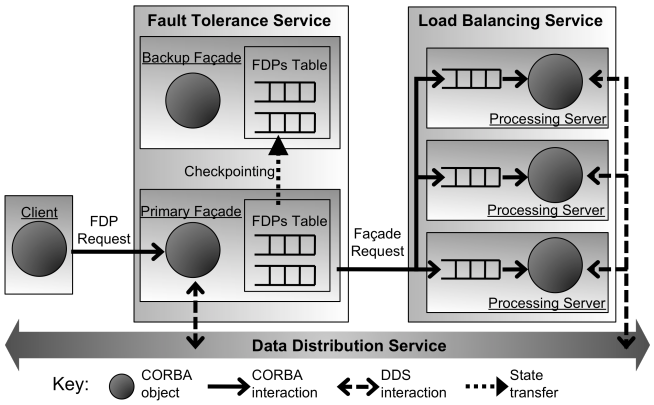

Fig. 4. A simplified view of the architecture of FDPS.

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 the clients.

The requests are relative to a specific flight track that is identified by means of an FDP-ID number: for each FDP-ID, the Façade dispatches no more than one request at time towards the PSs, and enqueues the others. The state of requests for each FDP is stored in a 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: FDP Tables are checkpointed at each update and transmitted to backup replicas, which are activated in the case of failure of the primary replica.

Since our aim is to test the fault-tolerance and load-balancing mechanisms, in this case study we include in the system model (and thus in the definition of the state) the number and type of requests in the FDP Tables, and the number of requests enqueued at each PS. The system model was not included in this paper due to space constraints; it is described in [32].

In our preliminary implementation, we considered only one service u_1 of the DS, the UPDATE interface, which is invoked by specifying the FDP-ID. For each FDP-ID, we composed the workload configuration with two parameters, $T_{m_{1,i}}$ and D_i: the first one specifies the period between two requests for the i-th FDP-ID; the second one imposes on PSs the time to spend in processing the respective UPDATE invocations. The Feeder interacts with FDPS through the middleware (Fig. 5). We fixed six FDP-IDs and set the domains for the parameters of the workload configuration ranging from 500ms to 5s, with a step of 500ms. In our tests, the application has been deployed on 100Mbps Ethernet LAN, using 3 Processing Servers, one active Façade and one backup Façade replica.

We conducted a preliminary experiment in order to evaluate the feasibility of the proposed approach. The Navigator was configured to drive the system in a

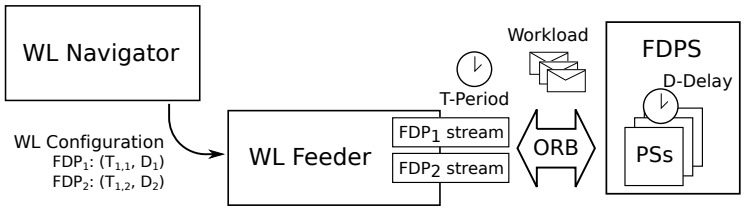

Fig. 5. The interaction among the WL Navigator, the WL Feeder and FDPS. The Feeder generates a *request stream* for each FDP-ID. Each request stream is configured through two parameters: a period between requests (T-Period), and a delay in processing of the requests associated with that stream (P-Delay).

state fulfilling the following conditions: (i) one PS is idle, the other two PSs are busy, (ii) there are requests enqueued by the middleware for the two busy PSs, and (iii) there are between 1 and 5 enqueued requests for each FDP Table. This scenario is interesting because it represents an hard-to-reach condition, as also discussed in [33]: there should be an idle Processing Server, while the other two PSs should be busy and have requests enqueued by the middleware for them (i.e., the enqueued requests should not be forwarded to the idle PS). This condition is actually possible since the round robin scheduler selects the PS for a request regardless of whether it is busy.

In the experiment, we aimed at reaching this target state with a high probability $p_{G,\tau}(\mathcal{R}_W)$ for at least 0.25s. The WG found a solution with $p_{\mathcal{S}_G,\tau}(\mathcal{R}_W) \backsimeq 66\%$ in the first population, after 30 minutes. At the fourth population of solutions and 2 hours, the best solution found by the WG was able to reach the target state with $p_{\mathcal{S}_G,\tau}(\mathcal{R}_W) = 100\%$.

6 Conclusion and future work

In this paper, we discussed an approach for state-driven workload generation in complex distributed systems. Our approach, based on a genetic algorithm, iteratively tunes the workload until a desired target state is reached. Our preliminary implementation on a real-world case study (a Flight Data Processing System) confirmed the feasibility of the approach and provided encouraging results. In future work, we will perform a more throughout evaluation of the approach, by evaluating fault tolerance mechanisms in the FDPS through fault injection. Moreover, we aim to further develop our implementation of the approach, in order to make it portable to other systems and to freely distribute it.

Acknowledgments

This work has been supported by the projects 'Embedded Systems in Critical Domains" (CUP B25B09000100007) within the framework of "POR Campania

FSE 2007-2013", and by the TENACE PRIN Project (n. 20103P34XC) funded by the Italian Ministry of Education, University and Research.

References

1. Chillarege, R., Iyer, R.: An experimental study of memory fault latency. IEEE Trans. Computers **38**(6) (1989) 869–874
2. Czeck, E., Siewiorek, D.: Observations on the Effects of Fault Manifestation as a Function of Workload. IEEE Trans. Computers **41**(5) (1992) 559–566
3. Meyer, J., Wei, L.: Analysis of workload influence on dependability. In: Proc. Intl. Fault-Tolerant Computing Symp. (1988) 84–89
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. 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
6. Vieira, M., Madeira, H.: A dependability benchmark for OLTP application environments. In: Proc. 29th Intl. Conf. on Very Large Data Bases. (2003) 742–753
7. 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
8. Meling, H., Montresor, A., Helvik, B., Babaoglu, O.: Jgroup/ARM: a distributed object group platform with autonomous replication management. Software: Practice and Experience **38**(9) (2008) 885–923
9. 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
10. Kiskis, D., Shin, K.: Swsl: A synthetic workload specification language for real-time systems. Software Engineering, IEEE Transactions on **20**(10) (1994) 798–811
11. Duraes, J., Madeira, H.: Generic faultloads based on software faults for dependability benchmarking. In: Dependable Systems and Networks, 2004 International Conference on, IEEE (2004) 285–294
12. Kalakech, A., Kanoun, K., Crouzet, Y., Arlat, J.: Benchmarking the Dependability of Windows NT4, 2000 and XP. In: Dependable Systems and Networks, 2004 International Conference on, IEEE (2004) 681–686
13. Zhang, F., Cheung, T.y.: Optimal transfer trees and distinguishing trees for testing observable nondeterministic finite-state machines. Software Engineering, IEEE Transactions on **29**(1) (2003) 1–14
14. Nachmanson, L., Veanes, M., Schulte, W., Tillmann, N., Grieskamp, W.: Optimal strategies for testing nondeterministic systems. ACM SIGSOFT Software Engineering Notes **29**(4) (2004) 55–64
15. McMinn, P.: Search-based software test data generation: a survey. Software Testing, Verification and Reliability **14**(2) (2004) 105–156
16. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT press (2000)
17. Holzmann, G.J., Smith, M.H.: An automated verification method for distributed systems software based on model extraction. Software Engineering, IEEE Transactions on **28**(4) (2002) 364–377
18. Kerbrat, A., Jéron, T., Groz, R.: Automated test generation from sdl specifications. In: The Next Millennium–Proceedings of the 9th SDL Forum. (1999) 135–152

19. Fernandez, J.C., Mounier, L., Pachon, C.: A model-based approach for robustness testing. Testing of Communicating Systems (2005) 313–313
20. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability (2012)
21. Jain, R.: The art of computer systems performance analysis. John Wiley & Sons New York (1991)
22. Avritzer, A., Weyuker, E.J.: Deriving workloads for performance testing. Software: Practice and Experience **26**(6) (1996) 613–633
23. Weyuker, E.J., Vokolos, F.I.: Experience with performance testing of software systems: issues, an approach, and case study. Software Engineering, IEEE Transactions on **26**(12) (2000) 1147–1156
24. Kiskis, D.L., Shin, K.G.: A synthetic workload for a distributed real-time system. Real-Time Systems **11**(1) (1996) 5–18
25. Arlat, J., Aguera, M., Crouzet, Y., Fabre, J., Martins, E., Powell, D.: Experimental evaluation of the fault tolerance of an atomic multicast system. Reliability, IEEE Transactions on **39**(4) (1990) 455–467
26. Basile, C., Wang, L., Kalbarczyk, Z., Iyer, R.: Group communication protocols under errors. In: Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on, IEEE (2003) 35–44
27. 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
28. Hoarau, W., Tixeuil, S.: A language-driven tool for fault injection in distributed systems. In: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing. (2005) 194–201
29. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed Systems: Concepts and Design. Addison-Wesley (2001)
30. 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
31. De Jong, K.A.: Evolutionary computation: a unified approach. MIT Press (2006)
32. Natella, R., Scippacercola, F.: State-Driven Workload Generation in Distributed Systems: System Model of an FDPS. Technical report (2013) Available at: `http://wpage.unina.it/roberto.natella/reports/`.
33. Natella, R., Cotroneo, D.: Emulation of transient software faults for dependability assessment: A case study. In: Proc. European Dependable Computing Conf. (2010) 23–32