Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

DICE testing tool – Initial version

Deliverable 5.4

Deliverable:	D5.4
Title:	DICE Testing Tools
Editor(s):	Matej Artač (XLAB)
Contributor(s):	Giuliano Casale (IMP), Tatiana Ustinova (IMP), Yifan Zhai (IMP), Matej Artač (XLAB)
Reviewers:	Youssef RIDENE (NETF), Richard Brown (ATC)
Type (R/DEM/DEC):	Demonstrator
Version:	1.0
Date:	31-January-2017
Status:	Initial version
Dissemination level:	Public
Download page:	http://www.dice-h2020.eu/deliverables/
Copyright:	Copyright © 2017, DICE consortium – All rights reserved

DICE partners

ATC:	Athens Technology Centre
FLEXI:	Flexiant Limited
IEAT:	Institutul E Austria Timisoara
IMP:	Imperial College of Science, Technology & Medicine
NETF:	Netfective Technology SA
PMI:	Politecnico di Milano
PRO:	Prodevelop SL
XLAB:	XLAB razvoj programske opreme in svetovanje d.o.o.
ZAR:	Universidad De Zaragoza

The DICE project (February 2015-January 2018) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644869

Executive summary

In this deliverable we present the initial release of the DICE Quality Testing (QT) tool with the description of the load injection methodology and corresponding tool implementation for testing Data-Intensive Applications (DIAs).

In this first release the load injection mechanism of the QT tool is designed for Apache Storm - a mature, stable and well-known Big Data technology for stream-based applications. The QT tool consists of two model. The first module is named QT-GEN and can generate input workload similar, but nevertheless non-identical, to the real workload data supplied to it, with prescribed rates for different types of messages. The tool then provides custom spouts for autonomic workload injection into DIAs, offered through a sub-module called QT-LIB, which is a Java library. Experiments are shown confirming the ability of the framework to stress performance and scalability in Storm-based DIAs.

Glossary

DIA	Data-Intensive Application
DICE	Data-Intensive Cloud Applications with iterative quality enhancements
D-Mon	DICE Monitoring Platform
MMAP	Marked Markovian Arrival Process
QoS	Quality of Service
QT	Quality Testing
QT-GEN	QT workload generator
QT-LIB	QT library
UML	Unified Modelling Language

Table of contents

List of Figures

List of Tables

1 Introduction

The DICE methodology proposes an iterative model for developing Data-Intensive Applications (DIAs), where applications are initially modelled using Unified Modelling Language (UML) diagrams, then prototyped and tested, sending the results in the form of feedback to the DIA developer. In this deliverable we present the first release of the DICE Quality Testing (QT) tool: a methodology and associated implementation for testing DIA prototypes.

The stated goal of the QT tool is to provide actuators for load injection into the DIA prototypes in order to assess Quality-of-Service (QoS) characteristics of the application such as performance, scalability and reliability. There is a variety of load testing tools available, both commercial [1],[2],[3],[4] and open-source [5], [6]. However, all these tools were developed for traditional web applications where incoming load is web traffic and users interact with the application's front end (website).

In the case of DIAs, incoming data (e.g. databases, streams) are the main workload drivers, but a recent DICE study of the QT market [7] revealed the lack of testing tools which could work with workloads of this kind. The goal of the QT tool is to provide the functionality of injecting stream workloads into a DIA and allowing the user to specify test scenarios.

1.1 Focus of this release: load injection for streaming workloads

Similar to other DICE tools, we have adopted for the development of the QT tool an incremental approach aimed at supported DICE technologies one-by-one. The current technologies on which DICE focuses are Apache Storm, Spark, Hadoop, Cassandra, and MongoDB. Big Data technologies and frameworks rapidly evolve, and since the deliverable D1.1 [8] the Cassandra and MongoDB databases became supported by the Apache JMeter [6] – a state-of-the-art load generation tool in the open source domain. MongoDB is supported natively, whereas Cassandra is supported through an external plugin [9]. There is also partial support for Apache Hadoop and Hbase [10], now available. This means that the research challenge associated with the load injection for these technologies is limited since a practical solution already exists, being JMeter the leading open source solution for stress testing.

Based on the above, we have decided to focus the QT tool scope into load injection capabilities for streaming workloads, which appear to have no advanced open source tool support yet. The QT tool presented in this deliverable therefore supports Apache Storm and we plan for the next tool release at M30 to develop an equivalent support for Apache Spark streaming.

1.2 Research challenges

Upon developing the QT tool we have identified the following research challenges (RC), which have been tackled by our initial research:

- **RC1: Load injection actuator design.** In traditional load injection, the workload submission relies on a client-server model [11] where external worker threads submit jobs to the application. This model is however *not* natural for platforms like Storm, where the message injection is done by dedicated software components, called *spouts*, that are part of the application itself, and which inject specific units of work packaged as messages that flow through the system. In systems of this kind, the load injection actuators should not be implemented as external elements, but rather as internal elements to the application.

Therefore there is a conceptual difference between developing QT tools for databases and similar tools for streaming platform. For example, being internal to the application itself, the resource usage of the workload generator in a streaming application will cumulate to the resource consumption of the application itself. This is not generally acceptable for database systems, but it's necessary in streaming applications, where the injection of messages consumes a separate pool of resources.

While also an *external* implementation may be considered, for example by providing a service equivalent to the ones used by Twitter to push data to external parties, a considerable advantage of the *internal* implementation is that the load injection actuator can have direct access to the data structures used within the DIA, including the custom message structure defined by the developer. This makes much simpler for the end user to define the streaming workload sent to the system using the same Java classes developed for the DIA itself and to automate the execution of the experiments. However, one important implication of this approach is that load injection needs to be provided as an embedded library that can be linked in the application code, with the developer defining the test workload through the IDE.

- **RC2: Scalability testing.** A requirement for any load injection tool is to be able to test the ability of a system to scale, i.e., to deliver Quality-of-Service (QoS) in terms of performance and scalability also when the parallelism level of the workload is increased (e.g., more users, greater data velocity, greater data volumes, etc.). It is therefore expected that a good implementation of the QT tools will be able to generate additional load as require by the end user, until observing a saturation of one or more resources. As shown later in the experimental results, this is possible with this release of QT.
- **RC3: Streaming workload generation.** Streaming-based DIAs typically focus on analysing incoming messages for specific topicsand trigger certain actions when pre-defined conditions are met on the payload of the messages. One research challenge is therefore to provide adequate tool support to make it easy for the user to generate time-varying workloads *and* time-varying message contents. Ideally, such workloads should resemble as much as possible the real workloads seen by the developer's company in production. For example, in the case of Twitter streaming data, one would like to generate workloads that are similar to ordinary Twitter feeds, but that are controllable in terms of arrival rate of individual kinds of messages (e.g., arrival rate of original tweets vs arrival rate of retweets, or arrival rates of messages from a specific user) and frequency of appearance of messages that trigger specific actions from the DIA. We have systematically investigated this problem throughout the research activity part of QT and we have developed a dedicated mathematical theory to address this problem. Due to the sophisticated nature of some of the equations involved, we point the interested reader to publication [12] for more details.

1.3 Research and Technical Achievements

In the details, the first release of the QT toolprovides an initial answer to the above research challenges through the following achievements:

Achievements pertaining RC1: Load injection actuator design. The QT tool is released as a Java library that can be used by any DIA developer to specify and inject workloads into an Apache Storm application. The QT tool is provided as a set of *custom spouts*, which allow the developer to specify

the intended behaviour of the workload during the load testing, defining, for example, constant-rate streams or linearly increasing injection rates.

We have found that this internal implementation of the load injection had also an importantadvantage: the deployment and execution of a load experiment can be done by simply deploying the application to production using the DICE Deployment Service.

Indeed, once deployed, the custom spouts autonomously start injecting the programmed workloads, carrying out a load injection experiment without user interaction or the need to setup a dedicated load injection testbed. This is made possible by the fact that platforms like Storm automate resource assignment to each spout, therefore essentially carrying out the task of configuring the resources for resource injection through spout without the need for user intervention.

Achievements pertaining RC2: Scalability testing. The QT tool's custom spouts are easily configurable to offer varying levels of parallelism in the injection of message streams. A simple configuration option is exposed to the developer to let him/her decide the threading level of the custom spouts and this will result in the proportional increase of the input rates of the messages. As such, provided that the deployment is performed on sufficient physical resources, it will always be possible to increase linearly message arrival rates and therefore drive the application resource to saturation. A software limitation exist on the maximum achievable rate by each spout (1ms between injections), but this did not appear to prove problematic in our validation tests.

Achievements pertaining RC3: Streaming workload generation. While the previous achievements are concerned primarily with implementation aspects, we have found that the definition of suitable workloads for testing Storm raised some novel fundamental mathematical questions that are worth investigating from a scientific standpoint.

It is commonplace in many workload generation tools for client-server systems to internally model the user navigation using some graph or automata, such as a Discrete-Time Markov Chain, which encodes the probability of the benchmark sending a request for a certain page or service. As discussed under the RC3 definition, for Storm we instead need to reason about arrival rates of different message classes, where a class is simply a message type defined according to some user criteria (e.g., tweets vs re-tweets).

In this release of the QT tool we have therefore developed a comprehensive mathematical theory of **multi-class arrival streams** and released a library of tools (implemented in MATLAB and called QT-GEN), based on what we termed M3A algorithms, to automate the fitting of empirical message traces into the mathematical models developed within this theory. Such models are **generative**, meaning that they can be simulated to generate a trace similar to the one used for fitting, but which is nevertheless non-identical. In this way, the QT tool's user can create a collection of random workloads with controlled arrival rates and burstiness of individual message types.

While rate modulation can be done in straightforward ways directly from the traces, e.g. by suitable message replication or dropping, what our mathematical theory delivers is the ability to preserve **a similar covariance structure** of the real trace, meaning that the **variability** and **burstiness** in the arrival stream rates observed in real traces will be similarly preserved in the artificial workloads generated by the QT tool.

Differently from modulation of the mean rates, control of the second-order properties related to variance and burstiness is notoriously difficult as it typically involve root-finding in a system of high-order nonlinear equations. However, the QT tool provides an extensive library of functions that greatly automate this process on behalf of the user, using heuristics and combining small-sized models into larger models with the ability to approximate rates, variability and burstiness of the trace.

A scientific publication documenting the theory underpinning the QT tool's workload generation was published in [12] on the INFORMS European Journal on Operations Research. The results of this paper have been packaged into two distributions of the same library, which consists of about 190 MATLAB functions: M3A-Toolbox (https://github.com/Imperial-AESOP/M3A) is a distribution intended for users not specifically interested to DICE, such as researchers and mathematicians. The same library then provided the mathematical backbone of QT-GEN, which delivers specialized functions to fit JSON workloads.

1.4 Deliverable organisation

The rest of this deliverable is organised as follows.

- In the Chapter 2 we review the QT tool requirements.
- In the Chapter 3 we provide an overview of the Apache Storm and its logging mechanism.
- In the Chapter 4 we discuss load generation and injection mechanisms of the QT tool.
- In the Chapter 5 we show some experimental results.

Source code, installation instructions and user documentation of the QT tool can be found in the DICE Knowledge Repository:

https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#quality

2 Requirements

In the Deliverable *D1.2 - Requirement specification*, released in the early phases of the DICE project [13], we presented the requirement analysis for the whole project, including the QT tool. This section provides an updated list of requirements, which considers our experience with application of the QT tool to practical scenarios. The actors involved are QTESTING_TOOLS, which represent the DICE Quality Testing tool, the CI_TOOLS, which are the DICE Continuous Integration tools, and the QA_TESTER, who is the developer or operator interested to validate the application quality.

2.1 'Must have' requirements

ID	R5.6
Title	Test workload generation
Priority	Must have
Description:	The QTESTING_TOOLS MUST be able to generate the workload with pre-specified characteristics for the APPLICATION.

ID	R5.8.2
Title	Starting the quality testing
Priority	Must have
Description:	The QTESTING_TOOLS MAY be invoked by the CI TOOLS or by the QA_TESTER

ID	R5.8.3
Title	Test run independence
Priority	Must have
Description:	The QTESTING_TOOLS MUST ensure that no side effects from past or ongoing tests leak into the runtime of any other test.

ID	R5.8.5
Title	Test outcome
Priority	Must have
Description:	The QTESTING_TOOLS MUST provide the test outcome to CI_TOOLS: success or failure

ID	R5.13
Title	Test the application for efficiency
Priority	Must have
Description:	The QTESTING_TOOLS MUST be capable of running tests with any configuration provided to it.

ID	R5.14.1
Title	Test the behaviour when resources become exhausted
Priority	Must have
Description:	The QTESTING_TOOLS MUST provide the ability to saturate and exhaust resources used by the application.

ID	R5.17
Title	Quick testing vs comprehensive testing
Priority	Must have
Description:	The QTESTING_TOOLS MUST receive as input parameter the scope of the tests to be run.

2.2 'Should have' requirements

ID	R5.7
Title	Data loading support
Priority	Should have
Description:	DEPLOYMENT_TOOLS and QTESTING_TOOLS SHOULD support bulk loading and bulk unloading of the data for the core building blocks.

ID	R5.14.2
Title	Trigger deliberate outages and problems to assess the application's behaviour under faults
Priority	Should have
Description:	The QTESTING_TOOLS SHOULD use the fault injection environments functionality to test the application's resilience.

ID	R5.7.2
Title	Data feed actuator
Priority	Should have
Description:	QTESTING_TOOLS SHOULD provide an actuator for sending generated or user-provided data to the application under test.

2.3 'Could have' requirements

ID	R5.15
Title	Test the application for safety
Priority	Could have
Description:	The QTESTING_TOOLS COULD test the application for safety properties.

2.4 QT tools and DICE Architecture

The positioning of the Quality Testing tools in the context of the DICE architecture is shown in Figure 1:

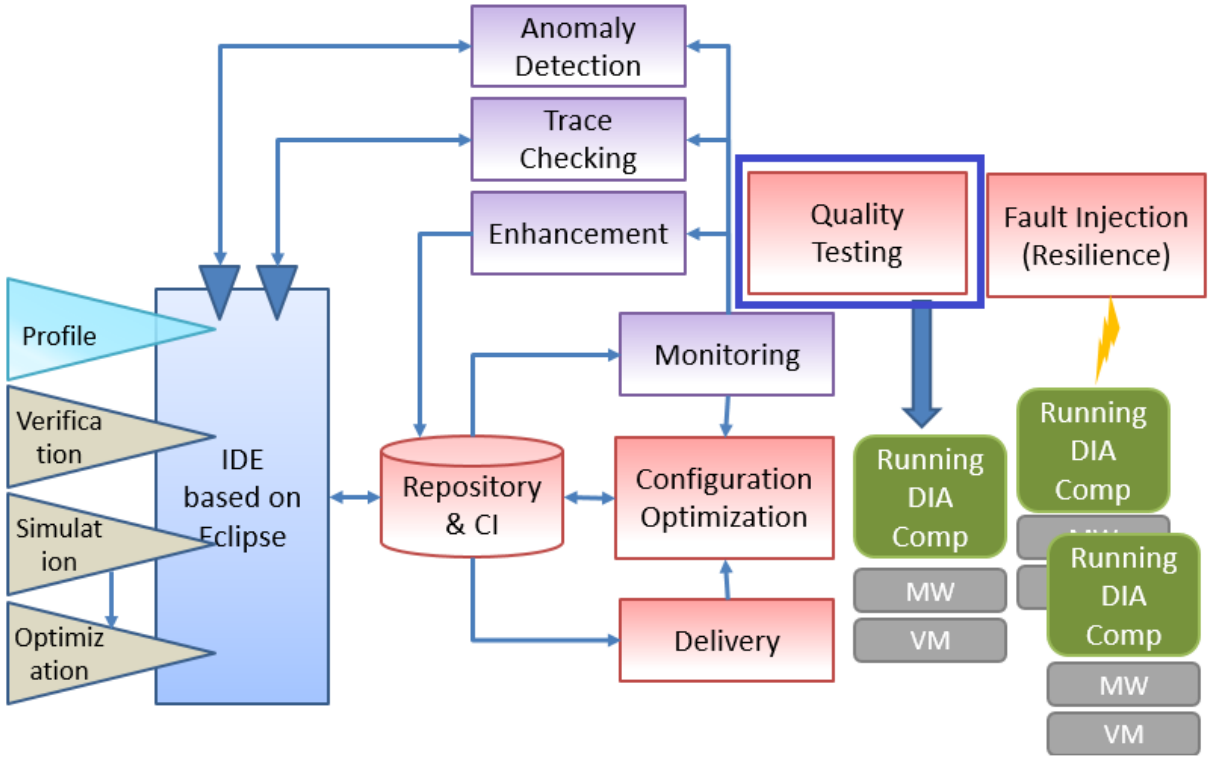

Figure 1 DICE Architecture - QT tools highlight

As shown in the figure, QT tools is a stand-alone component aimed at injecting load into specific running components of the data-intensive application. QT tools can be used in conjuction with the fault injection tools to assess the quality of the application in the presence of specific faults, such as simulated interruption of service for one or more VM, or saturated resources that compromise performance up to reaching unavailability of one or more services.

3 Storm overview: architecture, messages and logging system

The Quality Testing (QT) tool is designed to generate load and then inject it into a DIA built on Apache Storm. In the Storm-based application (also called a **topology**) the load injection point is called **spout**. A typical Storm topology consists of one or more spouts (data consumption points) and one or more bolts (functions that process received data). The QT tool is implemented in the form of custom spout which simulates load using a workload specification defined by the user. The standard topology spout is therefore replaced with the QT tool spout before application is deployed for testing, thus effectively making the QT tool a software component within the topology.

Throughout this deliverable the WordCount topology [14], which is included in the Apache Storm distribution [15] and Twitter data are adopted as **running examples** to illustrate the QT tool. However, the QT tool is general-purpose and can be applied to arbitrary Storm-based applications and arbitrary messages (other than tweets).

In order to understand how the QT tool has been developed and how it operates, it is important to understand the design and operation of the Apache Storm technology and Storm logs, which are discussed in the subsections below.

3.1 Apache Storm architecture: an overview

Figure 1 shows the workflow architecture of Apache Storm and an example of a typical Storm-based DIA. A working Storm cluster normally features one Nimbus (master node) and one or more Supervisors (working nodes) with the appropriate number of Zookeeper nodes acting as a coordinator of the supervisors and maintaining shared data with robust synchronisation techniques.

Once the topology has been instantiated, the Nimbus as the master node will first bootstrap it and analyse the total number of tasks (both for spout(s) and bolt(s)) of the topology. Then it will collect all the tasks, which are the Storm elements that perform the actual data processing. After that, Nimbus will evenly distribute the tasks to available Supervisors. Every supervisor runs one or more worker process, but it does not execute the tasks directly. Instead, it allocates the tasks to these worker processes, and each worker process will create the required amount of executors (operating system threads) to perform the task. While the number of tasks remains constant during execution of a topology, the number of threads that underpin the executors can be changed, subject to remaining less than the number of tasks at any given time.

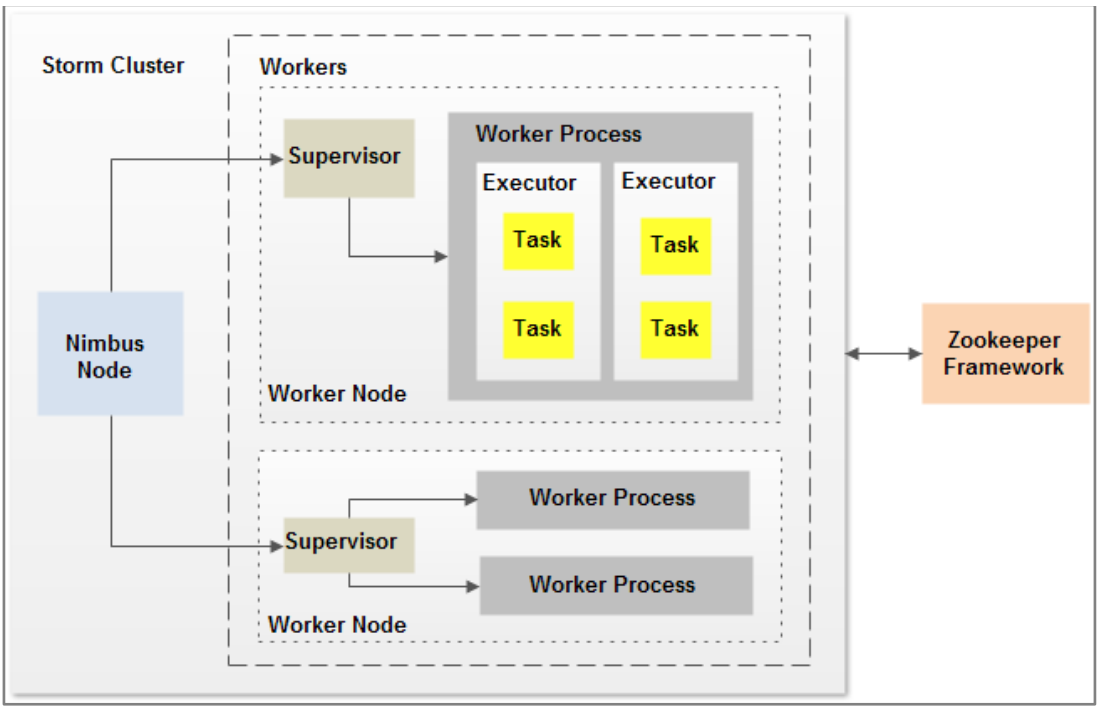

Figure 2. Apache Storm workflow architecture.

The Figure 2 shows the architecture of the WordCount topology as an example of a typical Storm-based DIA. The topology is represented by the Directed Acyclic Graph (DAG) and, by means of groupings, the incoming load is evenly distributed downstream to bolts. The groupings control data flow through the graph, and the topology may contain different type of groupings such as shuffle grouping, fields grouping, all grouping, global grouping, direct grouping, etc. The shuffle grouping is used in the Split Bolt of the WordCount topology. Such groupings are specific to WordCount and here merely serve the purpose of showing that the internal traffic behaviour of Storm-based DIAs depends on the application logic coded in spouts and bolts, which is difficult to anticipate before running experiments.

The QT tool can help to understand performance bottlenecks in this architecture, in particular when it comes to the bolts, which encompass the core elements of the application logic. That is, a working assumption of the Quality Testing is that the end user is interested in establishing the scalability of the application tasks processed by the bolts.

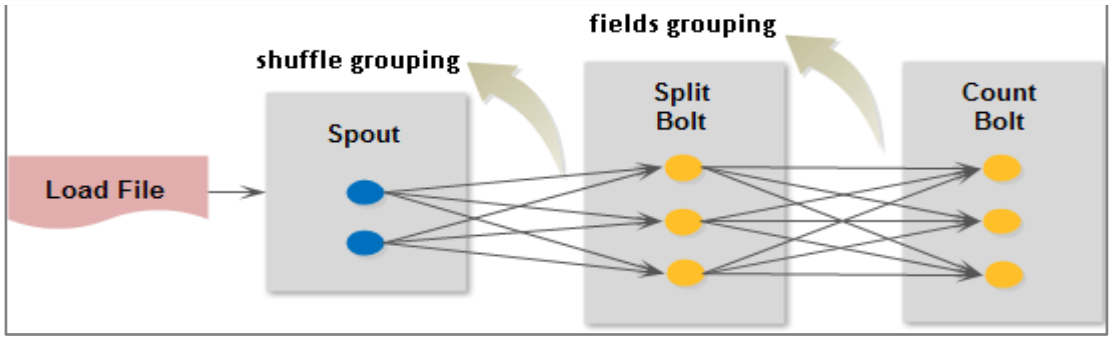

Figure 3. Apache Storm Dataflow Architecture.

3.2 Storm logs

Logs in the Apache Storm are responsible for tracking status, operations, error messages (e.g., ClassNotFoundException, IndexOutOfBoundException) and debugging information for all daemons. The status of each component within Storm (Nimbus, Supervisor(s), Zookeeper, Spout(s) and Bolt(s)) is monitored and recorded. Hence, log files contain a full report of each topology run. We now discuss the Storm logs, the types of logs and logs generation both in the Storm cluster and local modes, and, finally, describe the necessity and importance of log files for the QT tool workflow.

Apache Storm has two modes for running a topology: **cluster mode** and **local mode**. When running topology in the **cluster mode**, the whole Storm environment will call the `StormSubmitter` class to submit the topology. There is a field named "LOG" with the type of `static org.slf4j.Logger` in the `StormSubmitter` which uses Simple Logging Façade for Java (SLF4J) to record the logs. Once the topology is submitted, it will generate log files automatically and place them into the dedicated 'logs' folder in the Storm directory.

Once a topology is submitted, it will initially generate worker logs file and metrics logs file in the 'logs' directory. The worker logs are the main logs of Storm. Every record contains time stamp, daemon (executor), action information or status. Figure 3 shows the constituents of one record, and Figure 4 shows the example recording of different daemons in worker logs. Each worker has its own ID in the worker logs and the working processes among these workers would be cross-recorded, which means each LOG instance tracking in parallel without duplication of recording at the same time point.

The name of the worker logs file consists of 'worker' and its associated port number. Additionally, for each worker log, if the size of the records exceeds the maximum size of one log file, Storm will create a new file by extending the suffix of the log file, such as 'worker-6703.log.1', 'worker-6703.log.2', etc.

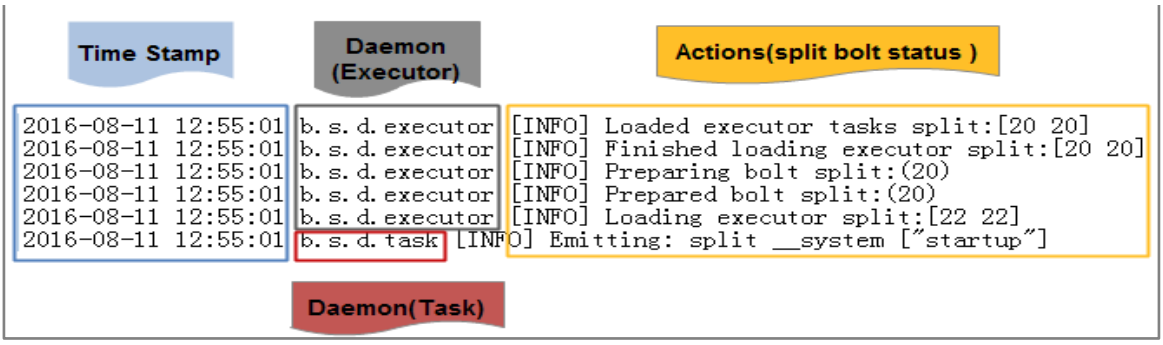

Figure 4. Constituents of one record in logs.

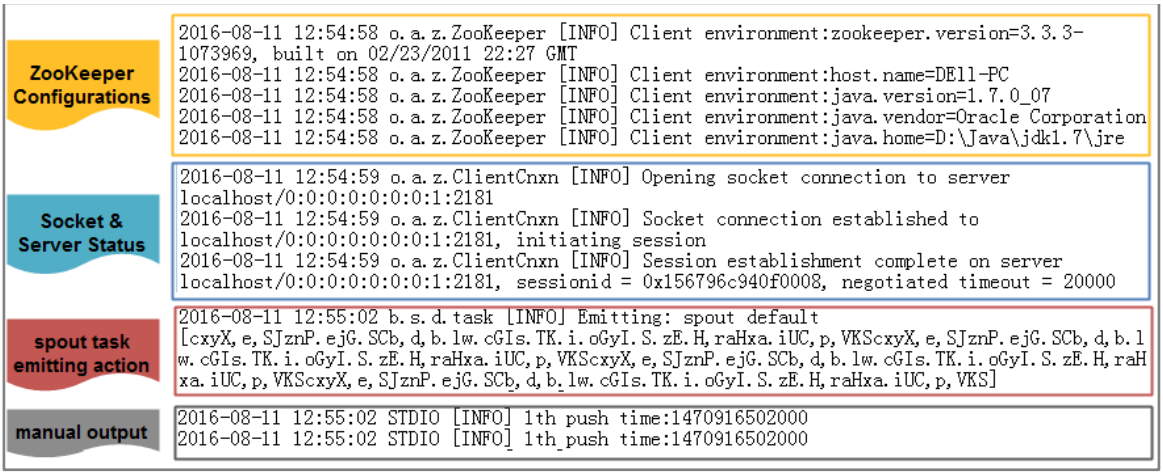

Figure 5. Example recording of different daemons in worker logs.

Metrics logs is a file which reports summary statistics across the full topology and tracks the digital information shown on the Nimbus UI console: counts of executes and acks (i.e., the 'acknowledgement' that indicates that the incoming piece of data was successfully executed by the specific bolt), average process latency per Bolt, worker heap usage, etc. Some of these metrics can be accessed directly through Storm IMetric interface [16].

Compared to the cluster mode, running a topology in the local mode does not actually start the Storm environment. Thus, there is no logs directory and the topology will not generate logs automatically. Instead, all the recorded messages will be displayed on the console. However, the logs can be manually dumped using the Simple Logging Façade for Java (SLF4J). In order to use the SLF4J, the following dependencies (shown in the Figure 5) need to be added to the `pom.xml` Maven project manager file. Once Maven successfully downloads the jar file with SLF4J, the `LoggerFactory` class can be used to record the information into the log files.

```
<dependency>
   <groupId>org.slf4j</groupId>
   <artifactId>slf4j-api</artifactId>
   <version>1.7.5</version>
</dependency>
<dependency>
   <groupId>org.slf4j</groupId>
   <artifactId>slf4j-simple</artifactId>
   <version>1.7.5</version>
   <scope>test</scope>
</dependency>
```

Figure 6. SLF4J dependencies in the *pom.xml* file for the local mode implementation.

4 Quality testing tool

In this part we introduce the main conceptual features of the quality testing tools, namely the ability to generate, extract from logs, and inject workloads into a Storm-based DIA in order to test its scalability.

4.1 Tool submodules

The QT tool is the combination of two independent contributions:

- QT-LIB: a Java library to define load injection experiments in Storm-based applications.
- QT-GEN: a tool for generation of traces to be used by QT-LIB.

QT-LIB is offered as a *jar file* that can be packaged in any Java-based Storm topology, providing the custom spouts for load injection and workload modulation. This tool can acquire external data to be pushed to the DIA either through MongoDB, for large datasets which can be exported as JSON files, or through external text files, as discussed in the next sections. The JSON file used uses a syntax compatible with MongoDB. An example of the reference JSON format is given later in this section for Twitter data. either based on Storm log data or on trace data provided in MongoDB-compliant JSON format.

Conversely, QT-GEN is offered in two forms: a compiled *binary file* for Windows and Linux, and a *sources distribution* for MATLAB users. QT-GEN's binaries can be run within the MATLAB Compiler Runtime, a royalty-free environment that is distributed for free and does not require to own a MATLAB license.

4.2 Overview of main features

4.2.1 Feature 1: automated load injection at a user-specified parallelism level

The QT tool offers a set of **custom spouts** that automate workload injection in the DIAs. The conceptual working of these spouts is simple: as soon as the DIA is deployed, these spouts automatically generate tuples using criteria that depend on the specific QT tool spout chosen by the developer. When the experiments reach a certain time duration, it completes and performance results are accessible to the user via the DICE monitoring platform (D-MON). Therefore, the injection process is entirely automated. Parameters such as the location of the data can be directly specified by the DIA developer in the constructor of the Spout.

One practical complication that arises in the use of such spouts is that in Storm spouts execute inside Java Virtual Machine (JVM) threads called Executors. An executor can only use a subset of the cores available on the computer that runs this JVM. However, the Spout class in Storm allows to configure the **parallelism hint** of spouts, a value that guides the platform to assign more resources to the spouts, although the way this is materially performed by the platform is scheduler-dependent. In our experiments, we have observed that increasing parallelism produces a linear increase in the message ingestion rate in the DIA, therefore allowing us to assess the DIA under heavy load conditions. For example, if *n* spouts are running in parallel, up to *n* times the maximal workload emitted by a single spout before core saturation may be emitted. Methodologically, the user does not need to know in advance the parallelism of the spout he needs to saturate the DIA. He can linearly increase the parallelism until finding that the output rate of the DIA does not grow anymore. This, together possibly with simultaneous saturation of resources, normally indicates that the

application has reached the scalability bottleneck. We envision the identification of this saturation point as the typical usage scenario for the QT tool.

4.2.2 Feature 2: fine-grained control over injection rates and timings

There are two ways to control the injection rates of messages using the QT tool. In the **uniform** scenario, the tool injects a batch of messages within a fixed timeslot into the system, for example every 1 second, and the user can specify in each second the injection rate within that slot. The figure below illustrates this scenario. The user can specify using the file shown on the left that for the first 10s, the injection rate will remain constant at 1040 bytes/s. The tool will ensure that this rate of tuple emission will be matched for all the timeslots that cover the first 10s of execution. Alternatively, as shown in the right, the user can provide a fine-grained specification of the data injection rate *for each* timeslot.

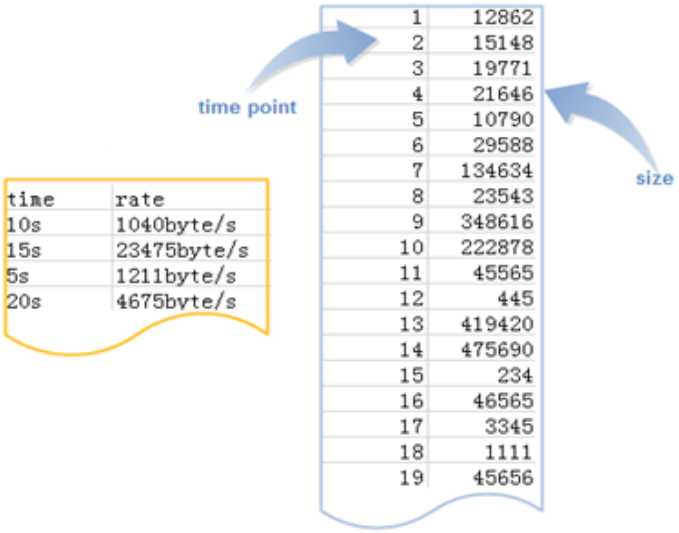

Figure 7. Specification of the data injection rates in the uniform scenario. The input files formats are supported, with aggregated (left) or detailed (right) specification of data injection rates or volumes.

Conversely, in the **exponential** scenario, the duration of each timeslot is randomized using a negative exponential distribution. If one assumes that in each slot the injection rate remains constant, this means that the tool will inject less data per time unit in longer timeslots, and more data in shorter timeslots. This therefore makes it possible to assess the response of the system under bursty conditions, which can lead for example to buffer overflows and data loss. A comparison of the two different timings is shown in the Figure below, where a 1-second uniform scenario is compared to an exponential scenario with randomized slot times.

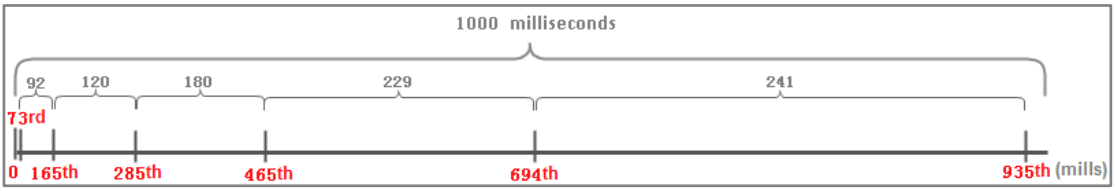

Figure 8. Example of the randomization based on the exponential distribution.

4.2.3 Feature 3: replay a nominal workload from log data

At the initial stages of application development it is important to ascertain that the system is stable while under the repeated application of identical 'nominal' workload over time. In order to execute this kind of test, the user should provide the file or stream with the data (load). However, this data is not always available, for example, Twitter feeds will never be the same. The QT tool is able in this case to parse an existing Storm log file and generate the required data stream to reproduce the tuples emitted during the execution of the nominal workload.

More in details, the QT tool extracts the data from the worker log files, leveraging the fact that tuples sent on the Storm spouts are recorded in the worker logs together with the timestamp of emission. Figure 8 illustrates the process of replaying the nominal workload for the exemplar WordCount topology. Figure 9 shows the content of a worker log file. The data that has been pushed by the Spout will be recorded by default into four fields (*timestamp, daemon, action, object)* and the value of the data.

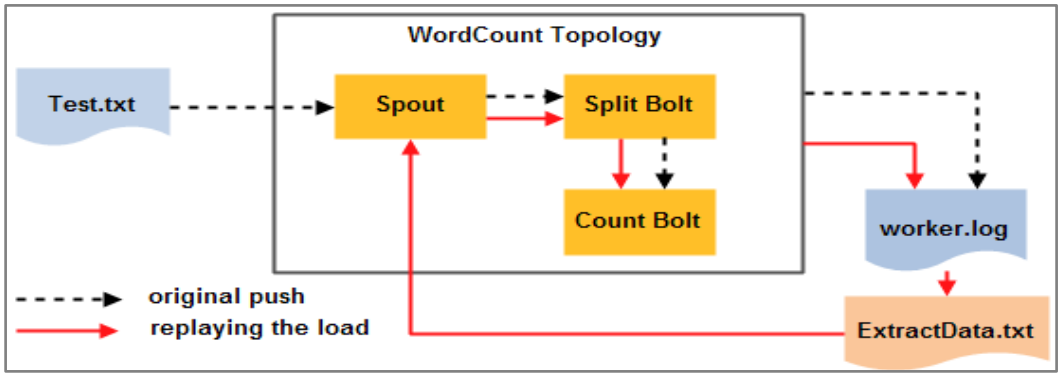

Figure 9: Algorithm of the 'replaying the load' feature of the QT tool for WordCount topology.

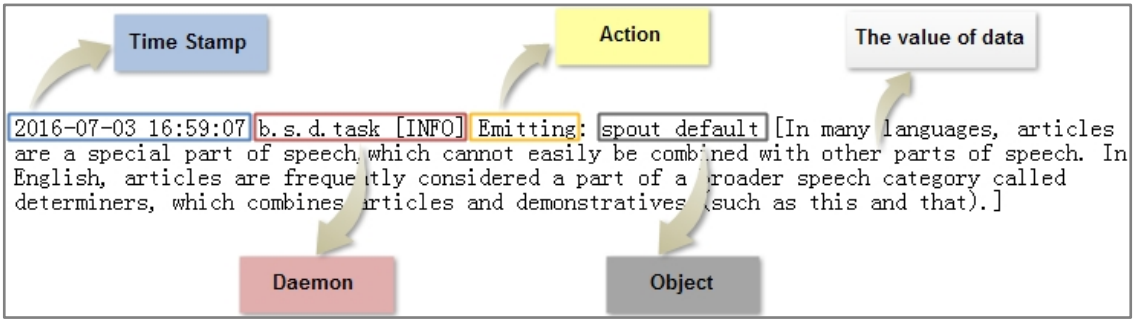

Figure 10. Example log data: the emitted spout tuple is stored in the log.

During the 1st injection of the nominal workload (original push), contained in the 'Test.txt' file, is sent to the platform which records the emitted tuples in the 'worker.log' file. The QT tool then parses 'worker.log' using user-customizable regular expressions and generates the 'ExtractData.txt' file containing the logged tuples and their timestamps. The difference between the information in the 'Test.txt' and the 'ExtractData.txt' is that the former contains the 'raw' text, while the latter contains the text in the format the spout has transformed it into (tuples) and in the order the spout has emitted these tuples to the bolts for processing. Therefore, since the same bolt used in the first experiment cannot be used in the second, a custom spout provided by QT needs to be deployed with the topology and the same tuples are injected again into? the system, replaying the same nominal load.

4.2.4 Feature 4: replay workload tuples from MongoDB

The QT tool is also able to replay tuples obtained from MongoDB. The user needs only to specify login credentials, Uniform Resource Identifier (URI) and position of the data within the database. Then the tuples are automatically extracted using Storm's *tuple()*, a generic conversion primitive applied to lists of generic java Objects, which in this setting are the selected contents of the databases. To do so, the QT tool modifies custom spouts developed by ATC as part of the SocialSensor project (http://socialsensor.eu/) as follows. The QT spout assumes that the MongoDB object includes a field with an injection timestamp. This name of this field can be customized indicated in the configuration XML of the QT tool. When the tool loads the *n*th Object from MongoDB it will compare the timestamp field with the one of the *(n-1)*th object and issue that tuple with a similar time lag. If the loading operation takes too long, the QT tool in a best effort simply issues the delayed tuple immediately.

4.2.5 Feature 5: generation of artificial workloads similar to a given workload trace

Lastly, we discuss the most advanced feature of QT, which is the ability to automatically generate artificial workloads from a given trace of Storm messages. The idea is to distinguish a set of message types within a given workload trace and produce a novel trace that has similar characteristics to the initial trace, without however being identical to it, and with arbitrary user-specified arrival rates.

QT assumes the availability of a JSON file where each entry represents a message to be serialized into tuples. For example, in collaboration with ATC in the context of their News&Media case study in DICE, we have considered the following JSON template for Twitter messages similar to the following (anonymized) tweet:

```
{ "className" :
"gr.iti.mklab.framework.abstractions.socialmedia.items.TwitterItem",
"reference" : "Twitter#00000000000000000",
"source" : "Twitter" ,
"title" : "@YYY: 4 DAYS TO GO #WorldCups",
"tags" : [ "WorldCups"],
"uid" : "Twitter#000000000",
"mentions" : [ "Twitter#111111111" , "Twitter#22222222222"],
"referencedUserId" : "Twitter#138372303",
"pageUrl" :
"https://twitter.com/CinemaMag/statuses/123456789123456789",
"links" : [ "http://fifa.to/xyxyxyxy"],
"publicationTime" : 1401310121000,
"insertionTime" : 0,
"mediaIds" : [ "Twitter#123456789123456789"],
"language" : "en",
"original" : false,
"likes" : 0,
"shares" : 0,
"comments" : 0,
"_id" : "Twitter#123456789123456789"}
```

In the real Twitter-based DIAs, a message like this is received immediately after publication, at 1 second intervals. The QT-GEN tool accepts in input a JSON file including an arbitrary long

sequence of such messages, parses it to extract the sequence of issue timestamp (in this example, the *"publicationTime")* and then generates a new sequence of messages where the messages are alternated in such a way to preserve a similar correlation among the user-defined types as seen in the original trace.

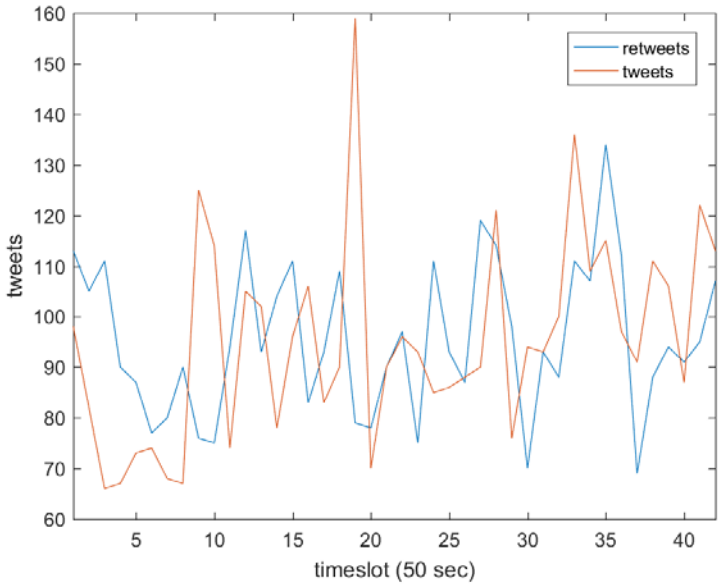

Figure 11. Correlation example: several peaks in tweets rates are followed by similar peaks in retweets rates. However, since the traffic is stochastic the matching is not perfect see, e.g., the tweets rate outlier at slot 18.

As a simple example, the QT tool can generate a new trace with:

- The same ratio of tweets and retweets as in the original trace, with the latter being identified by the "*original=false"* field in the above data structure.
- A statistically similar temporal spacing between tweets and retweets, so that volumes of tweets and retweets and the frequency and correlation of their peaks is reproduced also in the artificial workload.

The last property is illustrated in the Figure 10, which shows a correlated sequence of tweets and retweets in actual Twitter traces, where it is possible to note that, except for an outlier, most of the time a peak in tweet arrival rate is followed by a similar peak in retweets arrival rates, indicating correlation between the two streams. The QT tool is able to statistically analyse this correlation and reproduce it to the best possible extent in artificial workloads, using a class of traffic models called *Marked Markovian Arrival Processes* [12], which are a special class of Hidden Markov Models.

Markovian Arrival Processes (MAPs) are a class of stochastic processes used to model the arrivals from a traffic stream. *Marked MAPs* (MMAPs) are an extension of MAPs that allow to model arrivals of multiple types of traffic. The QT-GEN library provides a backend to the QT tool and it is designed for fitting marked traces with MMAPs and successive generation of representative traces. The latter problem is non-trivial due to the typically complex and nonlinear relationships between the statistical descriptors of the marked trace and the parameter of the MMAPs.

4.3 QT-LIB

4.3.1 Design approach

The QT tool offers a set of **custom spouts** that automate workload injection in the DIAs. The conceptual working of these spouts is simple: as soon as the DIA is deployed, these spouts

automatically generate tuples according to the user-specified workload. A **load generation interface** is available to the end user to specify an input data source (e.g., CSV, MongoDB); this interface provides the input tuples for the spout to inject load into the system and can be easily adapted to other databases or other textual or binary input sources.
Even though QT-LIB becomes part of the topology under test (both as a code and logical architecture), the system under test is a collection of Bolts that process data. Therefore, the Bolts and the relationships among them are embedded in the DIA code. QT-LIB provides the *uk.ic.dice.qt.** Java package, whose structure is presented in the Table 1:

Table 1: Structure of the *uk.ic.dice.qt.** Java package.

QT-LIB subtree	Description
uk.ic.dice.qt	Provides *QT-LIB*, a Spout factory that provides the entry point to QT-LIB.
uk.ic.dice.qt.logs	Auxiliary classes for load generation.
uk.ic.dice.qt.loadgen	Abstract interfaces for load generation.
uk.ic.dice.qt.loadgen.impl	Concrete implementation of load generation interfaces from external data sources (e.g., CSV, JSON, MongoDB).
uk.ic.dice.qt.spout	Custom spouts to replay load with controlled rate (RateSpout) or random data volumes (VolumeSpout).
uk.ic.dice.qt.util	Utilities, e.g., random number generation classes.

4.3.2 Usage example

We now illustrate the use of QT-LIB in combination with a toy example, the Exclamation topology from the storm-example directory [17]. Figure 11 illustrates the modified source code that includes invocation of QT-LIB. After initial application-specific configurations, the user creates the *QTLoadInjector* Spout factory and obtains a QT Spout to control the rate of injection of random textual data according to the rate specified in the CSV file. The user has also full control of the parallelism level of the load injection, using the standard controls for *parallelism hint* and *number of tasks* that are default in Storm.

```
public class ExclamationTopology {
  public static void main(String[] args) throws Exception {
    /* Storm topology builder */
    TopologyBuilder builder = new TopologyBuilder();

    /* Load custom property file */
    LoadConfigProperties loadconfigprop = LoadConfigProperties.createLoadConfigProperties();
    Properties props = loadconfigprop.getProps();
    Integer qtSpoutParallelism = loadconfigprop.getParallelism();
    Integer qtSpoutNumTasks = Integer.valueOf(props.getProperty("spout.task"));
    Map<String, Object> confSpout = loadconfigprop.getConfSpout();

    /* Create QT-LIB's Spout factory */
    QTLoadInjector qt = new QTLoadInjector();
    /* Obtain a spout to inject at prescribed rates specified in a CSV file */
    IRichSpout qtSpout = (IRichSpout) qt.getRateSpout("/home/user/exclamation/rateByCSV.csv");

    /* Install spout */
    builder.setSpout("word", qtSpout, qtSpoutParallelism).addConfigurations(confSpout).setNumTasks(qtSpoutNumTasks);
    builder.setBolt("exclaim1", new ExclamationBolt(), 1).shuffleGrouping("word").setNumTasks(1);
    builder.setBolt("exclaim2", new ExclamationBolt(),2).shuffleGrouping("exclaim1").setNumTasks(2);

    /* Configure and start topology */
    Config conf = new Config(); conf.setDebug(true); conf.setNumWorkers(1);
    StormSubmitter.submitTopology("qtspout", conf, builder.createTopology());
  }
}
```

Figure 12. Source code of the Exclamation topology modified for the invocation of QT-LIB.

After compiling the above DIA into a jar, it can be readily submitted to Storm an executed. Examples are shown later in the evaluation section (Section 5).

4.4 QT-GEN

In this section, we continued the presentation of the QT-GEN features illustrates in Section 4.2.5. We provide in particular technical details on the implementation of the workload generation.

4.4.1 A primer on Marked MAPs (MMAPs)

To produce artificial data, the QT-GEN tool resorts internally to a class of Hidden Markov Models called Marked Markovian Arrival Processes (MMAPs). The figure below illustrates a possible MMAP to describe a trace of messages of two types, “red” and “green”.

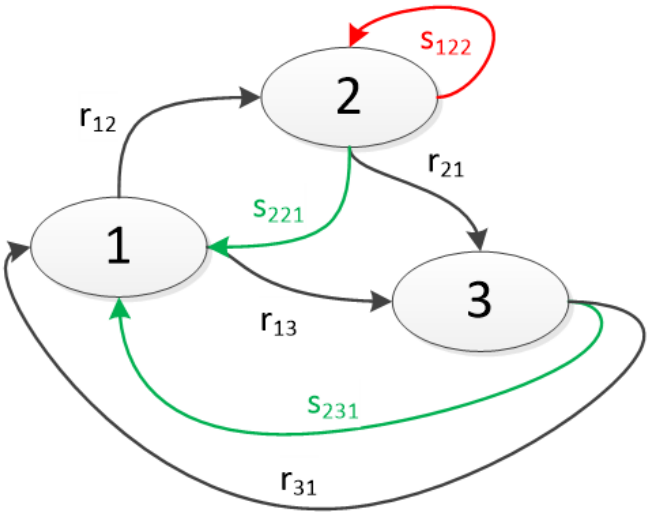

Figure 13. Example - a Marked MAP with 3 states and 2 markings (i.e., message types, here red and green).

Similarly to a Hidden Markov Model, the Marked MAP is a state-based model, where events are generated in each state according to a given distribution, which for the Marked MAP is an exponential distribution with parameters provided by the user. After spending an exponentially-distributed time in a given state, the Marked MAP jumps to other states at the rate specified on each transition, so that – in accordance with the standard rules of Markov chain simulation - a given transition is preferred with probability *proportional* to its rate. For example, transitions out of state 3 choose state one with 100% probability, but only with probability *p=s231/(r31+s231)* this transition will occur through the green arc. Upon traversing a black arc, nothing happens other than a change of the currently active state; for this reason this is called a *hidden transition*.
When the green or red arc are chosen during simulation of the Marked MAP, QT-GEN records the current simulation time and uses this value to define the time at which a message, either green or red, will be issued to Storm. In this way, if the holding times in each state and the transition rates are properly chosen one can shape the arrival traffic to the DIA. In statistics a *fitting algorithm* a procedure that accepts in input an empirical trace, such as a Storm log file, and produces in output a marked MAP with all model parameters chosen so that, upon simulating the marked MAP, this will have a high probability of producing a trace that is similar, in some sense, to the original trace. This fitting step is normally the most complex of the methodology, since the equations involved in the procedure are non-linear and difficult to solve. However, once the marked MAP is available two major features are provided to the user:

- As mentioned, the user can generate traces that are statistically similar to the original trace;
- The user can modify selected parameters in a simple manner, for example increasing as desired the rate or variability of the arrivals of just certain kind of messages.

Obtaining the same effect directly form the trace tends to be difficult, especially since correlations

may exist between events that are temporally far apart (e.g., as in weekly periodicities) that would be entirely ignored by randomly sampling the original trace. Vice versa, through marked MAPs we gain fine-grained control of the artificial workload generation.

4.4.2 Scientific highlights of the methodology

The scientific methodology focuses primarily on fitting trace data to the Marked MAPs, which from a mathematical standpoint is the most difficult step of the QT-GEN methodology. We point the interested user to [12] for a systematic treatment of the results obtained within DICE and we outlined in this section the following key results:

- QT-GEN generalizes the literature by defining a methodology to fit to traces Marked MAPs composed by an arbitrary number of states. Previous work has instead focussed on small Marked MAPs with 2 states.
- QT-GEN provides a "compression method" for Marked MAPs, called inter-position, that can dramatically reduce the number of states needed to fit complex traces, from hundreds, to normally less than ten. This comes at the price of sacrificing some control on the cross-correlation between message types, i.e., obtaining a less accurate fitting of the time-varying profile of the original trace.
- QT-GEN provides a comprehensive implementation that requires to the user only to specify the size of the Marked MAP, carrying out automatically all the mathematically-difficult decisions on behalf of the user.

4.4.3 QT-GEN functions

When run in the binary implementation, QT-GEN accepts the trace and generates automatically a new trace of desired length. However, in the sources distribution QT-GEN provides to the user a comprehensive library of about 80 functions, including the ability of fitting 7 different sub-types of Marked MAPs, which differ between each other for the number of states and specific mathematical properties, and a subset of functions dedicated to the calculation of statistical descriptors in time series with marked events (e.g., with different message types).
Since it is not possible to provide in this deliverable full details about this library of functions, we provide the brief description of the main groups and important functions in the Table 2:

Table 2: QT-GEN main groups and functions.

QT-GEN function	Description
m3afit_init	Initial loading and pre-processing for a trace with marked events
m3afit_auto	Automatically fit of a trace into a Marked MAP
m3afit_compress	Use the inter-position technique to reduce the Marked MAP size.
*m3pp2m_**	Fit two-state Marked MAPs with *m* marking types.
*m3ppnm_**	Fit *n*-state Marked MAPs with *m* marking types.

4.4.4 Usage example

We now illustrate the use of QT-GEN in the source modes. Details about the binary execution can be found on the tool wiki at the Github link provided in the next subsection. Figure 13 shows MATLAB code snippet, which determines 30 inter-arrival times for artificial tweets and their types.

```
tweets=loadjson('worldcup2014-5000.json');
%%
for i=1:length(tweets)
    ts(i)=tweets{i}.publicationTime; % timestamps
    orig(i)=tweets{i}.original; % tweet or retweet?
end
T = [ts(:),orig(:)]; T = sortrows(T,1); % sort by timestamp
mtrace = m3afit_init([0;diff(T(:,1))],T(:,2)); % determine inter-event times
M = m3afit_auto(mtrace,'NumStates',2); % fit the marked MAP
nSamples = 30; % determine 30 inter-arrival times of tweets and their types
[T,C]=mmap_sample(M,nSamples)
```

Figure 14. MATLAB function to determine inter-arrival times of artificial tweets and their types.

Each tweet is extracted from a JSON data structure identical to the one shown upon describing Feature 5 in Section 4.2.5. We assume here that the user has decided to generate artificial traces that reproduce the tweets and retweets' traffic. The user therefore stored in the *ts* vector the timestamps of arrival of the tweets, and their corresponding type is saved in the *orig* vector (1=tweet, 0=retweet). The user now initializes QT-GEN using the *m3afit_init* function and fits a 2-state Marked MAP on this data using the *m3afit_auto* function. The on-screen output of this call is shown in the Figure 14.

```
Rate: 0.001844
Fitting unified counting process...
Forcing exact rate: from 0.001844 to 0.001844
Rate: input = 0.002, output = 0.002
IDC(t1): input = 3.087, output = 3.087
IDC(t2): input = 3.087, output = 3.087
IDC(inf): input = 3.459, output = 3.459
M3(t2): input = 720.553, output = 72.598
Fitting per-class counting process...
Per-class fitting error: 394.178471
Rate class 1: input = 0.0008, output = 0.0008
Rate class 2: input = 0.0011, output = 0.0011
```

Figure 15. The example output of the call to the QT-GEN's *m3afit_auto* function.

Here, the *m3afit_auto* function has detected that the rate of arrival of tweets for this trace is *0.0008 tweets/ms* and the one of retweets is *0.0011 tweets/ms*, that the aggregated rate ignoring markings is *0.002 tweets/ms* and various statistics are given concerning other statistical properties (index of dispersion – IDC, 3-order moment of counts M3, see [12]). As indicated in the *output* fields, the generated MMAP upon simulation will generate events of the two types at the same *exact* rates of the original trace. At this point, the *mmap_sample* function enables the user to simulate the Marked MAP, obtaining in vector *T* this example 30 inter-event times between successive issues of messages and in vector *C* the indication whether these are tweets or retweets. This information is then used (not shown in the code snippet) to automatically generate the desired artificial traces by random sampling of the corresponding number of tweets and retweets from the *tweets* data structure and adjusting their timestamps according to the values in vector *T*. This procedure is fully automated in the binary release of QT-GEN.

The above example illustrates explicit use of the QT-GEN libraries internal functions that are supplied to advanced users. QT-GEN also comes with wrappers that hide the implementation complexity to the final user. For example, Figure 16 illustrates the same functionality of Figure 13

delivered using the wrapper function *qt_gen*, which abstracts the internal complexity of QT_GEN and it is applicable to arbitrary JSON datasets.

```
jsonFileName = 'worldcup2014-5000.json'; % 100 messages
nMessages = 30;

%% Example 1: generate 10 messages drawn from 2 classes (tweets or retweets)
newJSONFileName = 'worldcup2014-gen-2.json';
tsField = 'publicationTime'; % timestamp field
clField = 'original'; % class field
isNumeric = 1; % is the class field numeric?
qt_gen(jsonFileName,newJSONFileName,nMessages,tsField,clField,isNumeric);
```

Figure 16 QT-GEN wrapper for the example in Figure 13.

As shown in the figure the general syntax of the *qt_gen* function is as follows: JSON input and output file names are provided (*jsonFileName*, *newJSONFileName*), followed by an indication of the number of messages to be generated in the new trace. The *tsField* and *clField* attributes provide the names of the JSON data structure fields to be used for timestamps and class membership. The last parameter, *isNumeric*, specified if the *clField* is numeric (1) or a string (0). For example, if *clField* is the user id, the corresponding field (*uid*) is a string (e.g., "Twitter#255000000") and QT-GEN would treat each individual user messages as belonging to the same class. In this case, the workload generated by QT-GEN tries to mimic the real traffic produced by each individual Twitter users recorded in the trace.

QT-GEN requires installation of the MATLAB Compiler Runtime, as described in the Installation instructions. The tool depends on the KPC-Toolbox, maintained by the IMP unit, and JSONlab, a toolbox to encode-decode JSON for MATLAB. Both tools are freeware (MIT and BSD-3 licenses).

4.5 Integrated workflow

Figure 17 summarizes the integrated workflow of QT. The user can either generate load directly with QT-LIB, using a custom specified workload defined in the DIA, or generate a new workload using QT-GEN choosing to obtain this either from log files or from JSON datasets. The data produced by QT-GEN in output can be leveraged by QT's custom spouts for workload generation. Test results can be accessed using D-MON.

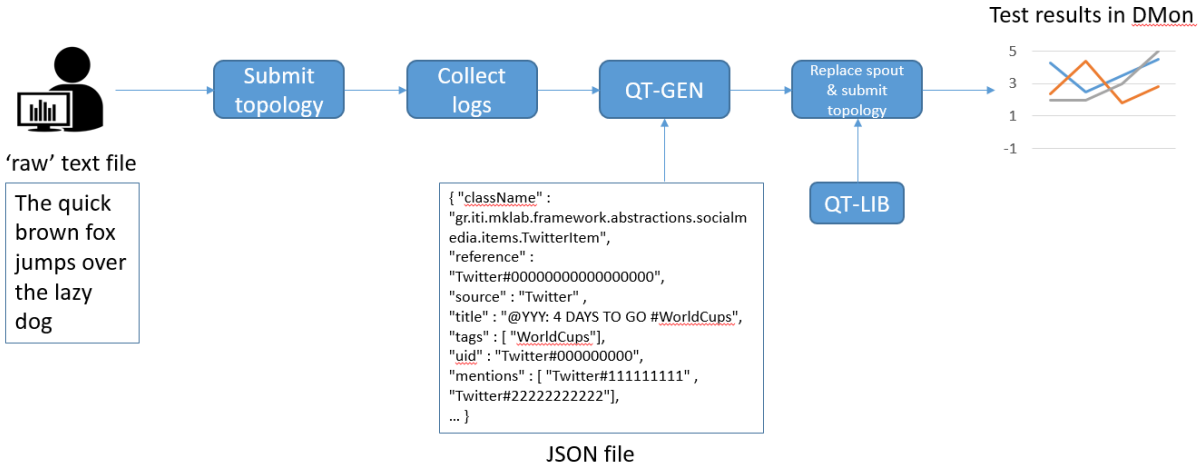

Figure 17 QT integrated workflow

5 Evaluation

In this section we present two examples of the DIA testing with the QT tool. The first example shows how varying the load with the QT spout (Feature 2, presented in the Section 4.2.2) influences DIA's data-processing time and presents the discussion of how these test results can be interpreted and used in the application's quality enhancement. The second example (Feature 1, presented in the Section 4.2.1) illustrates the ability of the QT framework to generate a higher rate of messages by increasing the parallelisation of the custom spouts.

5.1 Optimising application's latency with the timing of load injections

In this section we demonstrate the effect the uniform time interval division (the feature presented in the Section 4.2.2) has on the overall data processing time (complete latency) of the WordCount topology. Each of the incoming load rates (from the load scenario shown in the Figure 6, left) was distributed evenly (without the residuals) into the smaller bits within the second (by default, load injection into the topology is executed by the QT tool once per second). The summary of these splits is given in the Table 3:

Table 3: Uniform time division intervals.

Average uniform division			
UniformCut	Rate list, bytes/s	Uniform division list	Average uniform division
10	[1040, 23475, 1211, 4675]	[10, 5, 1, 5]	5.25
15	[1040, 23475, 1211, 4675]	[10, 5, 1, 5]	5.25
20	[1040, 23475, 1211, 4675]	[20, 5, 1, 5]	7.75
25	[1040, 23475, 1211, 4675]	[20, 25, 1, 25]	17.75
30	[1040, 23475, 1211, 4675]	[20, 25, 1, 25]	17.75
35	[1040, 23475, 1211, 4675]	[20, 25, 1, 25]	17.75
40	[1040, 23475, 1211, 4675]	[40, 25, 1, 25]	22.75
45	[1040, 23475, 1211, 4675]	[40, 25, 1, 25]	22.75

From the Table 3, it can be seen that each load volume per second should be divided into the intervals of different length (each load volume would still be split into even parts) to avoid residuals, so that the total amount of load pushed within the second would add up to the value set by the quality tester in the load scenario configuration file. For example, 1040 bytes/s when divided by 10 will result in 104 bytes being pushed every 100 ms. After 1 second the amount of bytes injected will total the 1040 bytes set by the quality tester. It gives the QT tool instruction into how many parts to split each load rate from the load scenario configuration file. For example (the first row of the Table 3), the quality tester sets 'UniformCut' value to 10 and provides the rates [1040, 23475, 1211, 4675] bytes/s. The QT tool will look into how many parts of equal size (up to 10) it could split each load rate into without producing residuals. The answer would be [10, 5, 1, 5] (on average 5.25). The effect of increasing the number of time sub-intervals on the overall topology data processing time (complete latency) is shown in the Figure 15:

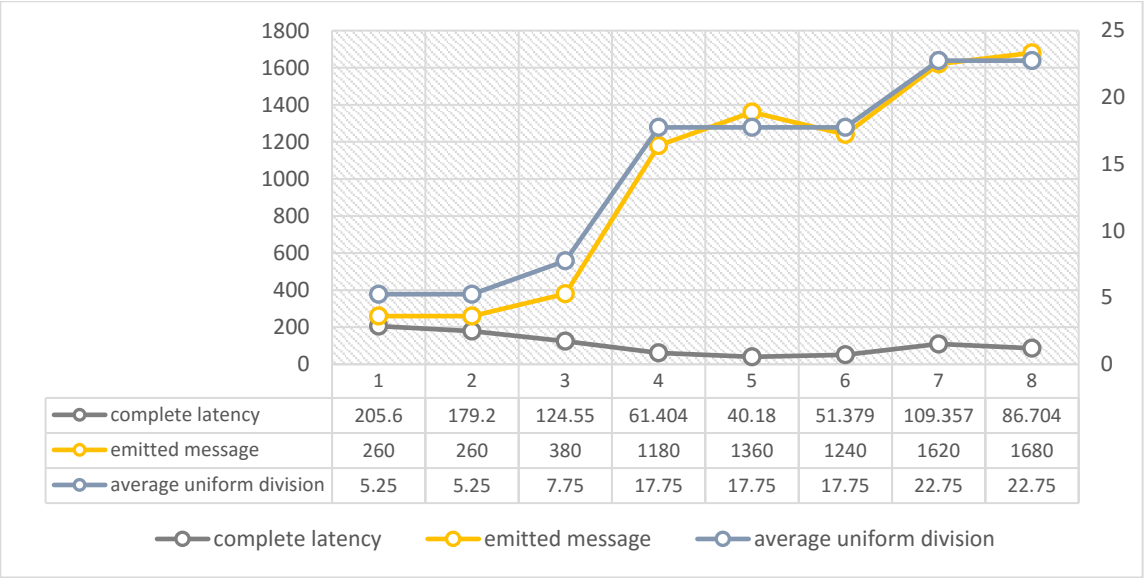

Figure 18. Effect of the load injection timing on the topology data processing time (complete latency).

Increasing time sub-intervals means that the smaller data bits are pushed into the topology, but more frequently within the second (the total number of load injection instances increases). This is illustrated on the Figure 15 by the blue and orange lines (average number of splits within second and total number of resulting data bits emitted into the topology respectively). Without any incoming load sub-division, when the load injection was done once a second, it took around 1000 s for the topology to process all injected data (in the load scenario presented in the Figure 6, left). From the Figure 15 we can see that splitting each load volume pushed within second into on average 5.25 parts reduces data processing time down to 200 seconds. Further splitting of the second (division of the data into even smaller bits, but which are pushed more frequently within the second) reduces complete latency even further (down to almost 40 seconds at the 17.75 average division) and then leads to it increasing back up to almost 87 seconds.

This downward and upward trend in data processing time can be explained. At first, when each load volume is pushed once a second, it takes a lot of time for the bolts to process these large chunks (especially 23,475 ones). When these chunks are split into smaller bits, it takes less time for the bolts to process them (even if they arrive to bolts more frequently). At some point the data size bits and their pushing rate come into balance with the data processing rate by the bolts (i.e. these data bits are processed with the same speed they arrive). This is the moment where complete latency drops to 40.2 s (total time to push data into the topology is 10 + 15 + 5 + 20 = 50 seconds into the load scenario employed in this example). 40 seconds is, obviously, lower than 50 (which is impossible: the overall data processing time cannot be lower than the time it takes to inject the load), but this can be attributed to the measurement error, as it is rather coarse (with the D-Mon the measurements would be more fine-grained).
When the load volumes are divided into even smaller bits (and pushed even more frequently within the second as a result), the data processing time starts increasing again (though not dramatically). This means that though the data chunks are even smaller and it takes less time for the bolts to process them, their number increases, which translates into the increased number of both the spout and the bolts tasks to be executed by Storm, with some of these tasks, probably, queueing before they are executed (in the current topology configuration).

By conducting the similar battery of tests with different ‘UniformCut’ values for their own load scenario, the quality tester can find the interval divisions that result in the optimal data processing

time by the topology (for the given application configuration settings). Translated to the real-life situation (in production), this means that the application would have optimal data processing time (under given configuration) if after reading the data from the source (stream or database), spout(s) push(es) it downstream to the bolt(s) for processing with the data size and frequency that correspond to the optimal value of 'UniformCut' found during the testing. In our example this value is 35.

The test results should be interpreted as follows (for our example): 1) We expect the load to arrive (in production) with the pattern 1040 bytes/s for the first 10 seconds, then with the rate of 23475 bytes/s for the next 15 seconds, 1211 bytes/s for the next 5 seconds and, finally, with 4675 bytes/s for the last 20 seconds (and then all over again). 2) Then, in order for the application to maintain the optimal data processing time, the application's spout (in our example there is one spout) should be able to distribute this load downstream to the bolts with the following pattern: 52 bytes/s every 50 ms (1040 bytes/20 and 1000 ms/20) for the first 10 seconds, 939 bytes/s every 40 ms (23475 bytes/25 and 1000 ms/25) for the next 15 seconds, 1211 bytes/s for the next 5 seconds (1211 is a natural number and cannot be divided by any number other than 1 without residual) and, finally, 187 bytes/s every 40 ms (4675 bytes/25 and 1000 ms/25) for the last 20 seconds.

5.2 Spout parallelism

Figure 16 illustrates the effect an increasing of spout parallelism parameter has on the overall number of messages spout reads from the data source within a given period and then sends to the bolts for processing.

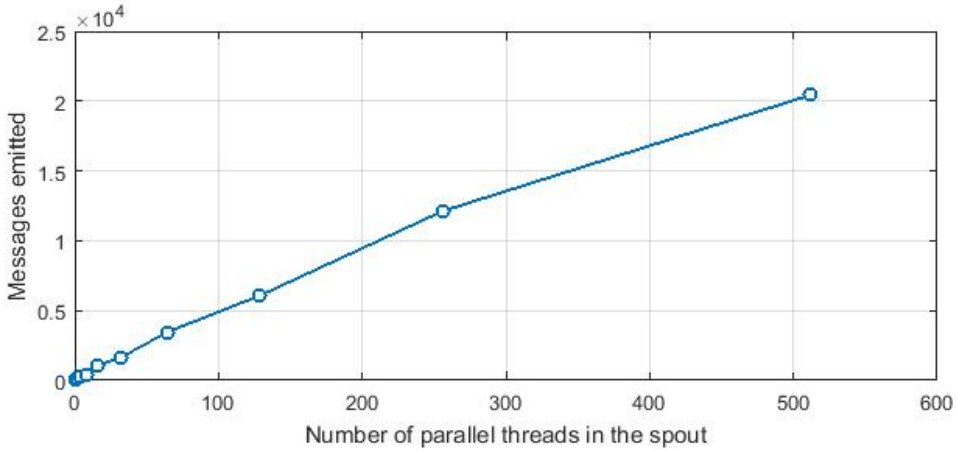

Figure 19. Spout parallelism.

As we can see from the figure above, as the number of parallel executors (threads) is increased in the QT's custom spout, a greater volume of emitted messages is produced within the platform. A roughly linear pattern is observed, suggesting that the additional capacity sized by the executors allows a proportional scaling of message volumes.

6 Conclusion

6.1 Summary of achievements

A summary of the main achievements of this first release of DICE QT is as follows:

- DICE QT allows load injection in Storm-based applications by means of custom spouts that can reproduce tabulated arrival rates of messages or inject in a controlled manner messages stored in a text file or in an external database, i.e., MongoDB.
- DICE QT can extract emitted tuples from Storm log files via regular expressions, therefore allowing the recording of a nominal workload that must be reproduced identically in future experiments.
- DICE QT can analyse a workload trace composed by different types of messages and fit the observed inter-arrival times of these messages into a special class of Hidden Markov Models, called MMAPs, from which new statistically similar traces can be generated in order to assess the system under varying load scenarios.

6.2 Fulfilment of requirements

In the Section 2 we provided a summary of the requirements. The Table 4 indicates the level that the DICE Quality Testing Tools comply in their initial release. The *Level of fulfilment* column has the following values:

- ✗ - not supported in the initial version yet
- ✓ - initial support
- ✓✓ - medium level support
- ✓✓✓ - fully supported

Table 4: Level of compliance of the initial version of the DICE delivery tools with the initial set of requirements.

Requirement	Title	Priority	Level of fulfilment
R5.6	Test workload generation	MUST	✓✓✓
R5.8.2	Starting the quality testing	MUST	✓✓✓
R5.8.3	Test run independence	MUST	✓✓✓
R5.8.5	Test outcome	MUST	✓
R5.13	Test the application for efficiency	MUST	✓✓✓
R5.14.1	Test the behaviour when resources become exhausted	MUST	✓✓
R5.17	Quick testing vs comprehensive testing	MUST	✓
R5.7	Data loading support	SHOULD	N/A in Storm
R5.7.2	Data feed actuator	SHOULD	✓
R5.14.2	Trigger deliberate outages and problems to assess the application's behaviour under faults	SHOULD	✗
R5.15	Test the application for safety	COULD	✗
R5.15.1	Test the application for data protection	COULD	✗

A discussion about the level of fulfilment for each requirement is as follows:

- R5.6 [MUST]: QTESTING_TOOLS is able to generate artificial workload for a Storm-based DIA using the specification provided by the QA_TESTER.
- R5.8.2 [MUST]: Since QTESTING_TOOLS is internally integrated to the application, the experiments are started by the QA_TESTER deploying the APPLICATION to production, a process which includes commit to CI_TOOLS.
- R5.8.3 [MUST]: Since QTESTING_TOOLS is internally integrated to the application, the user has full control over this aspect. Independent tests are obtained by simply updating the application deployment.
- R5.8.5 [MUST]: An integration plan with CI_TOOLS has been devised and reported, alongside the integration patterns for the other tools, in D1.4.
- R5.13 [MUST]: QTESTING_TOOLS is able to issue generate workloads to a Storm-based DIA.
- R5.14.1 [MUST]: QTESTING_TOOLS is able to increase the volume of message intake to the application until saturation and resource exhaustion is observed in the system.
- R5.17 [MUST]: QTESTING_TOOLS supports customization of the experiment duration.
- R5.7 [SHOULD]: In the context of Storm-based applications there is no near for in-memory loading of datasets as normally happens in load injection of databases (e.g., Cassandra). We will re-assess this requirement in the context of Spark, which will be the subject of M30 activities.
- R5.7.2 [SHOULD]: The MongoDB spout allows loading of messages from an external database.
- R5.14.2 [SHOULD]: The fault injection tools and the load injection tools are both released and at present entirely independent of each other, thus they can be easily combined for increased experiment realism. The do not appear to be entirely compatible for explicit cooperation since one is internally-integrated into the APPLICATION while the other is external to it.
- R5.15 [COULD]: At this time QTESTING_TOOLS is stress primarily performance and scalability characteristics. As mentioned above, there is no data persistently loaded in memory with Storm, it is therefore not possible with this technology to test a data protection scenario. ANOMALY_TRACE_TOOLS can be run on the resulting logs to check if any safety violation has incurred during the experiments.

6.3 Plan for M30 final release

A new release of DICE QT is planned for M30 and will focus on extending the tool to quality testing of Apache Spark streaming applications. The methodology presented here for Storm will be generalized and evaluated. Moreover, the tool will be integrated with the rest of the DICE platform, for example with the continuous integration toolchain. Detailed integration plans for each DICE tool are included in *Deliverable D1.4 – Architecture definition and integration plan – Final version*.

References

[1] Rational Performance Tester by IBM http://www-03.ibm.com/software/products/en/performance

[2] HP Load Runner http://www8.hp.com/uk/en/software-solutions/loadrunner-load-testing/

[3] Quali http://www.quali.com/cloudshell-platform/test-automation-authoring/

[4] Parasoft https://www.parasoft.com/wp-content/uploads/pdf/Parasoft_Continuous_Testing_DataSheet.pdf

[5] The Grinder http://grinder.sourceforge.net

[6] Apache JMeter http://jmeter.apache.org/index.html

[7] DICE Deliverable D7.6. Exploitation report – Intermediate version.

[8] DICE Deliverable D1.1. State-of-the-art analysis http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2015/08/D1.1_State-of-the-art-analysis1.pdf

[9] https://github.com/Netflix/CassJMeter/wiki

[10] https://jmeter-plugins.org/wiki/HadoopSet/

[11] Client-server model https://en.wikipedia.org/wiki/Client–server_model

[12] G. Casale et al., Compact Markov-modulated models for multiclass trace fitting. European Journal of Operational Research, 255:822–833, 2016.

[13] Deliverable D1.2 - Requirement specification. Available from: https://www.dice-h2020.eu/deliverables/

[14] WordCount topology explained http://admicloud.github.io/www/storm.html

[15] WordCount source code on GitHub https://github.com/apache/storm/blob/master/examples/storm-starter/src/jvm/org/apache/storm/starter/WordCountTopology.java

[16] Storm metrics http://storm.apache.org/releases/1.0.1/Metrics.html

[17] Apache Storm Exclamation topology https://github.com/apache/storm/blob/master/examples/storm-starter/src/jvm/org/apache/storm/starter/ExclamationTopology.java