MDload: a Model-Driven Workload Generator for Web Applications

Version 0.1

Last modified: October 16, 2014

Contents

1 Introduction

MDload is a model-driven workload generation tool that automatically generates requests to a web application by simulating a set of users. It has been developed as part of the MODAClouds Runtime Environment to fulfill the need for a self-adaptation testing tool, capable of injecting on-demand load to the cloud applications with characteristics of variability and burstiness. MDload is a general-purpose tool as it can be configured to generate traffic for different applications. This is achieved by implementing a set of routines that generate and submit application requests, allowing the user to define the specific requests to submit and their order. This therefore simplifies the testing of web applications, since it is only necessary to define how the user interacts with the application.

Once the user behavior has been specified, MDload emulates a set of users by relying on Selenium[1] to automate a Firefox web browser that sequentially submits requests to the application server. MDload is able to generate bursty traffic, dynamically changing the traffic mix, that is, the proportion of requests types generated by the users. This behavior better represents the actual workload faced by the application.

[1] http://docs.seleniumhq.org/

The operation of MDLOAD is built in three logical layers. The lowest layer is the *request* level, which considers atomic requests submitted by the user. The second layer is the *session* level, where a single user submits a sequence of semantically-correct requests, reproducing an expected usage pattern. The highest logical level is the *benchmark* level, where a number of sessions are managed to achieve the predefined workload characteristics. These levels are associated with the three main actors participating in the traffic generation, namely the Overseer, the Dispatcher and the UserAgents, which are implemented as Java Threads. The main thread spawns an instance of the Overseer, which is the object delegated to coordinate the test at the benchmark level. The Overseer simply creates an instance of a Dispatcher, which creates and manages UserAgents threads. Each UserAgent generates traffic by submitting requests to the Selenium driver that produces HTTP requests via Firefox. This operation is therefore automated by MDLOAD, and it is only necessary to specify the user behavior as described next.

MDLOAD is composed of three main libraries: `mdload`, `mdload-dev`, and `mdload-matlab`. The dependencies are shown in Figure 1. Using these libraries, the MDLOAD user can specify the user behavior for a particular application an application-specific library. This library will then be used by MDLOAD to generate the application workload. The next section provides a step-by-step guide on how this can be achieved, using an example application.

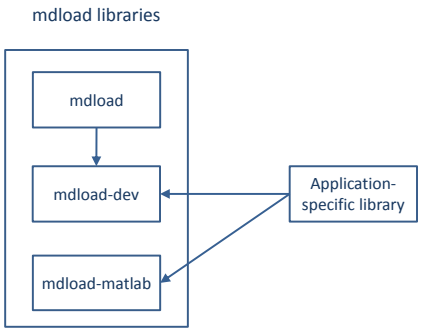

Figure 1: MDLOAD components

1.1 Specifying User Behavior

To specify the application user behavior, the MDLOAD user needs to specify (extend) two main (abstract) classes: `Request` and `Session`. These abstract classes are defined in the `mdload.client.workload` package, distributed as part of the `mdload-dev` project. The class `Request` is used to determine the action necessary to submit each of the possible application requests, which is done by implementing the `action` method. Thus, one `Request` class needs to be implemented for each type of application request to generate. The `action` method has Selenium `Web driver` object as its only parameter, which enables the action to use its functionality to interact with the application, e.g., clicking, clearing and filling fills, and submitting forms, among others.

The second abstract class to extend as part of MDLOAD is `Session`, which determines how an application user submits the different requests defined as `Request` classes. The constructor of a `Session` object includes a unique user identifier that enables MDLOAD to distinguish among the different users active in a benchmark. The MDLOAD user needs to implement two methods, `getWarmUp` and `getNext`, both of which provide a set of `Requests` to execute, in the form of a Linked List. The `getWarmUp` method is used during a first stage called *Warm Up*, which is typically used to register users with the application, so that they are able to access it later on. The `getWarmUp` method therefore returns a list of `Requests` that reflect this behavior, including for instance clicks to registration forms, filling and submitting these forms, among others. The `getNext` method is used once the *Warm Up* period has finished, and it therefore returns a

list of requests that reflect the common usage of the application, including both logged-in and non-logged in users.

1.2 Automatic generation of requests from Palladio models

Palladio component models provide a compact specification of a software application, including the workload that the users are expected to generate. To take advantage of this, MDload offers the module `mdload-matlab`, which generates a sequence of requests according to the usage defined in the Palladio model. The module is developed in Matlab[2] and uses as input an XML file containing a Layered Queueing Network (LQN) model. This XML file can be easily obtained using the PCM2LQN transformation implemented in Palladio Bench, and therefore can help in bringing to the runtime environment the Palladio models.

[2]This module is implemented in Matlab, but its binaries do not require a Matlab license to be executed.

2 Installation and Quick Start

To start using MDload, download the libraries by cloning the repositories publicly available at

- `mdload`:
  `https://github.com/imperial-modaclouds/modaclouds-mdload`
- `mdload-dev`:
  `https://github.com/imperial-modaclouds/modaclouds-mdload-dev`
- `mdload-ofbiz`:
  `https://github.com/imperial-modaclouds/modaclouds-mdload-ofbiz`
- `mdload-matlab`:
  `https://github.com/imperial-modaclouds/modaclouds-mdload-matlab`

2.1 Example: Quick start with OFBiz

The fastest way to start using MDload is to make use of the implementation already provided for the OFBiz e-commerce application. To do this,

1. Download, install, and deploy the latest release of Apache OFBiz 12 from `https://ofbiz.apache.org/download.html`. After deployment, OFBiz is available on the URL `BASE_URL`, and listening on port `BASE_PORT`, which normally is 8080. This means that OFBiz can be accessed on
   `OFBIZ_URL:OFBIZ_PORT/ecommerce/`
2. Download Firefox 13, which is compatible with the workload generator, and is available from `https://ftp.mozilla.org/pub/mozilla.org/firefox/releases`
3. Import the projects `mdload`, `mdload-dev`, and `mdload-ofbiz`, into your favorite Java IDE. Add `mdload-dev` as a dependency of `mdload` and `mdload-ofbiz`.
4. In the `mdload-ofbiz` project, modify the `config.userdefs` to specify the properties
   - `BASE_URL`: URL to access OFBiz, e.g., http://myofbiz.com
   - `BASE_PORT`: port where OFBiz accepts requests, normally 8080.
5. Create a `jar` file from the project `mdload-ofbiz`, say it is called `ofbiz.jar`.
6. In the `mdload` project, modify the `config.client` file to specify the properties
   - `WORKLOAD_JAR`: full path to the jar file created in the previous step, e.g., in a Windows platform this could be `C://path//to//my//jar//ofbiz.jar`. Notice that MDload is a Java application, and is therefore portable among different platforms.
   - `WORKLOAD_CLASS`: location of the main session class within the above-mentioned jar. In this case it is `mdload.userdefined.session.Test`
7. In the `mdload` project, Execute the class `MDLoad`. This should start a Firefox browser, or as many as indicated in the `TOTAL_USERS` property in the `config.client` file, and start emulating the interaction of these users with the application.

2.2 Example: Generating traffic from a Palladio/LQN model

MDload also allows generating the application workload from a Palladio Component Model (PCM). A PCM is a compact description of an application, its components, the resources where it is deployed, and the workload if faces. PCMs can be built and modified with the Palladio Bench tool, available for download on `http://www.palladio-simulator.com/tools/download/`. More details on PCM can be found in [1]. As part of the `mdload-matlab` repository, a *sample PCM* is provided in the file `data/OfBizBranch.zip`, which is a model of the OFBiz application.

Once a PCM model is available, it is necessary to generate the associated Layered Queueing Network (LQN) model. This can be achieved in Palladio Bench, which implements the PCM2LQN transformation introduced in [3]. LQNs are a popular abstraction for application performance models [2]. The `mdload-matlab` repository also provides a *sample LQN* in the file `data/ofbizBranch.xml`.

You can thus use this LQN model for workload generation, following the next steps.

1. After downloading all the repositories mentioned at the beginning of this section, follow steps 1 to 4 in the previous sub-section.

2. Download and install MATLAB Compiler Runtime (MCR) version 2012b, freely available on `http://www.mathworks.co.uk/products/compiler/mcr/`.

3. Start the MDL server. Go to your local copy of the `mdload-matlab` repository. In the folder `test` you will find the `MDL` executable file for Windows, together with a properties file named `MDL.properties`. Double click on the `MDL.bat` file, which runs `MDL.exe` with `MDL.properties` as parameter. This will start the MDL server that parses the LQN model and returns a set of requests to execute by the application client.

4. In the `mdload-ofbiz` project, modify the `config.userdefs` file to set the following properties

   - `LQN_FILE`: set this to the sample LQN model provided, e.g., in a Windows platform this could be `C://path//to//mdload-matlab//data//ofbizBranch.xml`. Notice that MDload is a Java application, and is therefore portable among different platforms.
   - `LQN_SERVER`: set this to the IP address where the MDL server is running, e.g., if it is your local server set it to `localhost`.
   - `LQN_PORT`: set this to the port where the MDL server is listening. This should be the same as the `port` property in the `MDL.properties` file. The default value is 6350.

5. Create a `jar` file from the project `mdload-ofbiz`, say it is called `ofbiz.jar`.

6. In the `mdload` project, modify the `config.client` file to specify the properties

   - `WORKLOAD_JAR`: full path to the jar file created in the previous step, e.g., in a Windows platform this could be `C://path//to//my//jar//ofbiz.jar`. Notice that MDload is a Java application, and is therefore portable among different platforms.
   - `WORKLOAD_CLASS`: location of the main session class within the above-mentioned jar. In this case it is `mdload.userdefined.session.TestMatlab`

7. Execute the class `MDLoad`. This should start a Firefox browser, or as many as indicated in the `TOTAL_USERS` property in the `config.client` file, and start emulating the interaction of these users with the application. In the command-line window where the MDL server executes, you can also see the log of request sequences generated from the LQN model, and passed to the workload generator.

3 User's guide

In this section we illustrate how to develop the classes needed to generate the workload for an specific application, relying on the MDLOAD tool. We will use the OFBiz example, the classes of which are provided in the `mdload-ofbiz` repository.

3.1 Workload generation

Since MDLOAD takes care of all the details of the workload generation, it is enough to focus on describing the user behavior in terms of sessions and requests. A request is an atomic interaction of the user with the application, typically accessing a service URL. A session is a set of consecutive requests submitted by a single user, describing a behavior necessary to achieve a certain service. For instance, a user registering for the first time with the application submits a different sets of requests than a user that already has an account.

As shown in the next code snippet, when developing a workload generator with MDLOAD one must define a class that extends the `Session` abstract class. This involves providing two main methods: `getWarmUp` and `getNext`. These two methods return a list of requests to be executed by the user. The `getWarmUp` method is used during the first stage of a benchmark, called warm-up period, where basic information necessary for the normal operation of the application can be collected. For instance, users could register in this phase to populate the users database. The `getNext` method is used during the main stage of the benchmark, called steady state, which comes after the warm up period, and represents the expected behavior of the application users.

```
public class Test extends Session {
   public LinkedList<Request> getWarmup() {
   ...
   }

   public LinkedList<Request> getNext() {
   ...
   }
}
```

The actual implementation of the `getWarmUp` method is provided in the next snippet. Here we observe the creation of a `List` if `Requests` to return, and the addition of each of the specific requests. In particular we observe that the `RegisterDetails` request requires the user details as a parameter in the constructor. To this end, we define the class `mdload.userdefined.UserLoginDetails`, which holds the details of a specific user, and the `Test` class has a field of this class, thus associating an individual user to each session.

```
public LinkedList<Request> getWarmup() {
      LinkedList<Request> session = new LinkedList<Request>();
      session.add( new StartSession());
      session.add( new Home() );
      session.add( new Register() );
      session.add( new RegisterDetails(user) );
      return session;
   }
```

Similarly, the `getNext` method is defined in the following snippet, returning a different sequence of requests to execute. We notice that the `LoginDetail` request also makes use of the user details mentioned above.

```
public LinkedList<Request> getNext() {
```

```
        LinkedList<Request> session = new LinkedList<Request>();
        session.add( new StartSession());
        session.add( new Home() );
        session.add( new Login() );
        session.add( new LoginDetails(user) );
        session.add( new QuickAddMain() );
        session.add( new Main() );
        session.add( new Logout() );
        session.add( new EndSession());
        return session;
    }
```

The previous definition of the `Test` class, which extends the `Session` class, is provided in the `mdload.userdefined.session` package. We now illustrate the definition of the `Request` classes, which are provided in the `mdload.userdefined.request` package. The next snippet exemplifies the `Home` request, which is the basic request that accesses the application home page. This class extends the `Request` abstract class, for which it implements the `action` method. This method receives a Selenium `WebDriver`, which it uses to access the application homepage. In this case the URL and PORT to access the application are provided by the utility class `UserDefs`. This method also returns the response time experienced in processing this request.

```
public class Home extends Request
{
    public long action( WebDriver driver){
        long in = System.currentTimeMillis();
        driver.get(UserDefs.BASE_URL + ":" + UserDefs.BASE_PORT + ...
            "/ecommerce/");
        return System.currentTimeMillis() - in;
    }
}
```

A different example is given in the next snippet, where the action `action` method of the `Login` request uses the driver to find a link with the text "Login", and clicks it to initiate the user login.

```
public class Login extends Request
{
    public long action( WebDriver driver ) {
        long in = System.currentTimeMillis();
        driver.findElement(By.linkText("Login")).click();
        return System.currentTimeMillis() - in;
    }
}
```

After the previous request, the user submits the `LoginDetails` request, which provides the required login details and clicks the submit button. In this case the `LoginDetails` class requires a `UserLoginDetails` field to keep the details of the specific user.

```
public class LoginDetails extends Request
{
    private UserLoginDetails user;

    public LoginDetails(UserLoginDetails u)
    {
        super();
        user = u;
    }
```

```
    public long action( WebDriver driver ) {
        long in = System.currentTimeMillis();
        driver.findElement(By.id("userName")).clear();
        driver.findElement(By.id("userName")).sendKeys(user.getUsername());
        driver.findElement(By.id("password")).clear();
        driver.findElement(By.id("password")).sendKeys(user.getPassword());
        driver.findElement(By.cssSelector("input.button")).click();
        return System.currentTimeMillis() - in ;
    }
}
```

3.2 Workload generation from a Palladio/LQN model

As described in sections 1 and 2, MDLOAD provides the means to generate the application workload using a Palladio Component Model (PCM) or a Layered Queueing Network (LQN) model. This is illustrated in the `TestMatlab` class, which is provided in the `mdload-ofbiz` repository. To understand the operation of this class, we must first mention that MDLOAD comes with a component called the MDL server, distributed in the `mdload-matlab` repository. This component operates as a server, receiving instructions to generate a set of requests from an LQN model. The operation of MDLOAD in this case thus relies on first starting the MDL server, and then connecting to it to obtain a set of requests.

This operation is illustrated in the following snippet, where the `getNext` method of the `TestMatlab` class is depicted. The first step consists of connecting to the MDL server. Next, this class submits the command "NEXT", after which the MDL server sends a sequence of requests back. This list of requests, separated by semicolon, is parsed, creating a `Request` object for each request using the `injectNewRequest` method. This method uses the requests classes defined in the package `mdload.userdefined.request`, described in the previous section. These `Request` objects are added to the list returned by the method.

```
public LinkedList<Request> getNext() {
    LinkedList<Request> session = new LinkedList<Request>();
    connected = connect();
    if (connected){
        try {
            //request list from MDL service
            String command;
            //request next request
            command = "NEXT";
            out.println(command);
            out.flush();
            //response
            String resp = in.readLine();
            System.out.println(resp);
            // tokenize response
            String[] parts = resp.split(":");
            // fill session linkedlist with response
            for(int i = 0; i < parts.length; i++){
                session.add(injectNewRequest(parts[i]));
            }
            //close connection;
            command = "CLOSE";
            out.println(command);
            out.flush();
            out.close();
            in.close();
            lineSocket.close();
        }catch(IOException e){
            System.err.println("An I/O exception occurred:"+e.getStackTrace());
```

```
            System.exit(1);
        }
    }
    return session;
}
```

A challenge with PCM and LQN models is that their level of granularity may be too coarse to directly generate requests from the application calls defined in these models. To overcome this, we have implemented a simple method in MATLAB, currently provided as part of the `MDLprotocol` script in the `mdload-matlab` repository. As depicted in the snippet below, this method parses each application call in the PCM model into a valid application request. For instance, the LQN model, which is also provided in the repository, includes a call to a service named "checkLogin". This is not an actual application request, and instead represents two consecutive requests: `Login` and `LoginDetails`. This method therefore maps each application call in the PCM/LQN model into an appropriate set of requests, that can be readily used by the workload generator to emulate the user behavior. This of course is application and model specific, and needs to be dealt with in each case. Thus, the user needs to modify this method accordingly for the specific application and model considered.

```
function reqName = parseCallRequest(callName)
    reqName = '';
    if length(callName) >= 26 && strcmpi( callName(1:24), ...
        'RequestHandler_HandlerIF')
        reqName = strtok(callName(26:end), '_');
        switch reqName
            case 'main'
                reqName = 'Main';
            case 'checkLogin'
                reqName = 'Login:LoginDetails';
            case 'quickadd'
                reqName = 'QuickAddMain';
            case 'addcartbulk'
                reqName = 'CartAddAll';
            case 'checkoutoptions'
                reqName = 'CheckOut';
            case 'processorder'
                reqName = 'OrderHistory';
            case 'orderhistory'
                reqName = 'OrderHistory';
            case 'orderstatus'
                reqName = 'CartView';
            case 'logout'
                reqName = 'Logout';
        end

    end

end
```

A Configuration properties - config.client

The following are the configuration properties to specify in the config.client file, in the mdload project.

CACHING: enables the browser cache options if set to 1; disables them if set to 0. Default value: 0.

CLIENT_MASTER_IP: IP address of the client node that acts as the master node.

DISPATCHER_PORT: port where the Overseer sends messages to the Dispatcher regarding the status of the benchmark. Default value: 9878.

EXECUTION_TIME_MS: total execution time (in milliseconds) of the benchmark. Default value: 60000.

OVERSEER_PORT: port where the Dispatcher sends messages to the Overseer regarding the status of the benchmark. Default value: 9877.

PAGELOAD_TIMEOUT_MS: timeout (in milliseconds) when loading a page with Selenium. Default value: 60000.

IMPLICIT_WAIT_MS: implicit wait time (in milliseconds) for the Selenium driver. Default value: 600000.

RANDOM_SEED: seed used for random number generation. Used to generate the inter-session arrival times, and the inter-request think times. Default value: 0L.

SESSION_IAT_ACF_GEOMDECAY_RATE: decay rate of the auto-correlation function of the stochastic process that generates the session inter-arrival times. Default value: 0.0

SESSION_IAT_MEAN_MS: mean session inter-arrival time in milliseconds. After a session is complete, it undergoes an inter-arrival time before a new session is created. Default value: 1000.

SESSION_IAT_STDEV_MS: standard deviation of the session inter-arrival times in milliseconds. Default value: 1000.

THINK_TIME_ACF_GEOMDECAY_RATE: decay rate of the auto-correlation function of the stochastic process that generates the inter-request think times. Default value: 0.

THINK_TIME_MEAN_MS: mean inter-request think time in milliseconds. This think time is the time elapsed between two successive requests within a session. Default value: 1000.

THINK_TIME_STDEV_MS: standard deviation of the inter-request think times in milliseconds. Default value: 1000.

TOTAL_USERS: total number of application users emulated. Default value: 1.

WORKLOAD_JAR: full path of the JAR file with the workload definitions specific for the application.

WORKLOAD_CLASS: main class within the workload JAR file. This class extends the Session class.

B Brief class description

This appendix briefly describes the classes of the MDLOAD tool.

B.1 Package `mdload.client`

ClientDefs: utility class with static fields that hold the configuration properties defined in the `client.config` file. It also loads the `client.config` file and updates the corresponsding fields.

MDLoad: main class of the package, uses the configuration information from `ClientDefs`, defines the distributions for the think times and the session inter-arrival times (IATs). Creates and launches the `Overseer`.

Overseer: setups a `Dispatcher` with given think time and inter-session inter-arrival distributions. Keeps track of the signals submitted by the `Dispatcher` to record the beginning and the end of the three main stages of the benchmark: warm-up, steady state, and cool-down.

Dispatcher: creates the `UserAgents`, which emulate the web application users. Upon creation each `UserAgent` is assigned a Selenium driver for a Firefox browser to interact with the application. This object keeps a dispatch queue, where the `UserAgents` add themselves when they are ready to be assigned a new set of requests. The set of requests is obtained from the class `WORKLOAD_CLASS` defined within the JAR file `WORKLOAD_JAR`, by means of two methods: `getWarmUp` and `getNext`. These provide the sequence of requests to be executed during the WarmUp and the Steady State periods.

UserAgent: this is the class that emulates the behavior of the application user. Its main task is to execute the sequence of requests assigned to it by the `Dispatcher`, with interleaving think times.

B.2 Package `mdload.client.util`

This package provides support classes for workload generation.

Distribution: abstract class that defines the method `next`, which returns a random number according to a probability distribution.

DMPH: defines a marked PH process which is used to represent the state of the workload generator. It is used to model burstiness in the arrival process.

B.3 Package `mdload.client.comm`

This package provides support classes for communication between the main MDLOAD components.

InChannel: abstract class that defines the method `receive`, in charge of receiving signals via an application socket.

OutChannel: abstract class that defines the method `send`, in charge of receiving signals via an application socket.

TCPInChannel: extends the `InChannel` class using Java Sockets and Server Sockets.

TCPOutChannel: extends the `OutChannel` class using Java Sockets.

UDPInChannel: extends the `InChannel` class using Java Datagram Sockets.

UDPOutChannel: extends the `OutChannel` class using Java Datagram Sockets.

Signal: defines a set of signals for communication between the MDload components.

C Example: configuration properties in the OFBiz example

The following are the configuration properties to specify in the `config.userDefs` file, specific for the OFBiz application.

BASE_URL: URL of the OFBiz application.

BASE_PORT: port to access the OFBiz application.

LQN_FILE: full path of the LQN model from which the workload must be generated, in XML format.

LQN_SERVER: IP address where the MDL server is running, e.g., if it is your local server set it to `localhost`.

LQN_PORT: port where the MDL server is listening. Default value: 6350.

References

[1] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-based performance prediction with the palladio component model. In *Proceedings of the 6th WOSP*, pages 54–65, 2007.

[2] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *Software Engineering, IEEE Transactions on*, 35(2):148–161, 2009.

[3] Heiko Koziolek and Ralf Reussner. A model transformation from the palladio component model to layered queueing networks. In *Performance Evaluation: Metrics, Models and Benchmarks*, volume 5119 of *Lecture Notes in Computer Science*, pages 58–78. Springer, 2008.