Flood - User Guide

Kajetan Rzepecki

September 25, 2013

Contents

1 Introduction

Flood is a load simulator useful for automatic **Comet/PUSH application** stress-testing. It is **asynchronous, event based** and enables you to create JSON encoded **test scenarios of arbitrary complexity** involving **tens of thousands of simulated users**, no Erlang required!

1.1 Use cases

Some of the most common use cases that **Flood** might be helpful in testing are:

- **Massive, real-time, on-line chats**,
- **Publisher-Subscriber channels**,
- **Instant messaging**.

However, Flood is general enough to test *any* event-based Comet application that uses the supported protocols.

1.2 Supported Protocols

Flood currently supports the **Socket.IO** protocol over **WebSocket** and **XHR-polling** transports with emphasis on Socket.IO event based communication. Flood also has *some* capabilities of using **raw HTTP** requests.

1.3 Dependencies

Flood uses several awesome libraries that are listed below. Since Flood is currently in development, *no particular stable versions are required* and by default the newest available versions are pulled in.

- Ibrowse - an HTTP client, found here.
- Lager - a logging framework, found here.
- Folsom - a metrics system, found here.
- JSONx - a fast JSON parser, found here.
- Jesse - a JSON Schema validator, found here.
- `websocket_client` - a WebSocket client, found here.

2 Inner workings

This section describes what happens behind the scenes in **Flood** and how it reflects its usage.

2.1 Simulated Users

- FSMs
- State transitions

2.2 User sessions

2.2.1 Session selection

- Roulette algorithm

2.2.2 Session inheritance

- Single inheritance ordering.
- Multiple inheritance ordering.
- Why so OOP?

2.2.3 Actions & Event handlers

- $\text{on}_{\text{socketio}}$
- on_{event}
- $\text{on}_{\text{timeout}}$

2.2.4 Timers & Counters

- Starting/stopping/restarting timers
- Managing counters

2.3 Flood phases

- Phases purpose
- Phase goals

3 Flood scenarios

This section describes the Flood scenario files and gives some general guildelines for writting them. Example scenarios can be found here.

3.1 Scenario file

Flood uses JSON to encode test scenarios, no Erlang is required. Each scenario resides in a separate file and optionally several goal files (described in detail later). The overall structure of a Flood scenario consists of three required sections:

```
{
    "server" : {
        // Server setup.
    },

    "phases" : {
        // Test phases & goals.

        "phase_I" : {
            ...
        },
        ...
    },

    "sessions" : {
        // User session descriptions.

        "session_A" : {
            ...
        },
        ...
    }
}
```

3.2 Server setup

The `server` section is rather straightforward; it is used to setup the server connection. It has to define several mandatory fields:

```
"server" : {
    "host" : "",     // The server host.
    "port" : 0,      // The server post.
    "endpoint" : "", // Endpoint used to connect to.
    "metadata" : {}  // Server-wide metadata (optional).
}
```

Example server configuration that will cause Flood to connect to http://localhost:80/socket.io/1/ and define some server-wide metadata (more on metadata can be found here):

```
"server" : {
    "host" : "localhost",
    "port" : 80,
    "endpoint" : "/socket.io/1/",
    "metadata" : {
        "foo" : "bar"
    }
}
```

3.3 Phases setup

The `phases` section may define several arbitrarily named Flood phases. The ordering does not matter, as each phase explicitly names its start time.

```
"phases" : {
    "A" : {
        // A's description.
    },

    "B" : {
        // B's description.
    },
    ...
}
```

Each phase description has to follow this format:

```
"phase_I" : {
    "users" : 0,          // Number of users spawned during this phase.
    "user_sessions" : [], // Sessions spawned users should follow.

    "start_time" : 0,     // Time (in milliseconds) at which to start this phase.
    "spawn_duration" : 0, // Duration (in milliseconds) Flood should take to spawn the users.

    "goal" : {},          // Goal of this phase (optional).
    "test_interval" : 0,  // Interval (in milliseconds) of the goal checks (optional).
    "timeout" : 0,        // Timeout (in milliseconds) of this phase (optional).

    "metadata" : {}       // Phase-wide metadata (optional).
}
```

The meaning of each of the fields is as follows:

- `users` - an integer number of users spawned during this phase. It is **mandatory**.
- `user_sessions` - a array of Flood user session names; the concrete user session will be selected at **random according to a sessions weight** (more about this can be found here). It is **mandatory**.
- `start_time` - an integer value that names a point in time (**in milliseconds**), relative to the start of the Flood, at which a phase should be started. It is **mandatory**.
- `spawn_duration` - an integer value that tells Flood how much time (**in milliseconds**) it should take to spawn `users` number of users. Users are spawned uniformly throughout this duration. Keep in mind that for various performance related reasons Flood **may actually take longer** to spawn the users, however it will never take less time to do so. This field is **mandatory**.
- `goal` - either an arbitrary JSON term that is a description of the goal of this phase (more on goals can be found here) or a string containing a path to the file containing the goal description relative to scenario file. This field is **optional**; not defining it will result in no goal checking whatsoever.
- `test_interval` - an integer value that tells Flood at what intervals (**in milliseconds**) in should check whether the `goal` has been reached. It is **optional**; not defining it will result in a single check at the phase `timeout`.
- `timeout` - an integer value that names a point in time (**in milliseconds**), relative to the start of the Flood, at which a phase should be terminated if it is still running. It is **optional**.
- `metadata` - a JSON object defining some phase-wide metadata (more on metadata later). It is **optional**.

Example `phases` setup:

```
"phases" : {
    "phase_I" : {
        "metadata" : { },

        "users" : 1000,
        "user_sessions" : ["session_A", "session_B"],

        "start_time" : 1000,
```

```
        "spawn_duration" : 1000
    },

    "phase_II" : {
        "metadata" : { },

        "users" : 1000,
        "user_sessions" : ["session_C"],

        "start_time" : 2000,
        "spawn_duration" : 5000

        "goal" : "./goal.jsonschema",
        "test_interval" : 100,
        "timeout" : 10000
    }
}
```

This setup will schedule two Flood phases. The first phase, `phase_I`, will start at 1000 ms and spawn 1000 users following either `session_A` or `session_B` over 1000 ms duration. The second phase, `phase_II`, will start at 2000 ms and spawn 1000 users following `session_C` over 5000 ms duration. Additionally, a `phase_II` goal check will be scheduled every 100 ms starting at 2000 ms and running util the goal provided in "./goal.jsonschema" file is met or until the phase timeout, set at 10000 ms, is reached.

3.4 User session setup

The `sessions` section may define several arbitrarily named Flood user sessions. The ordering does not matter, as each session explicitly names its relations to other sessions.

```
"sessions" : {
    "session_A" : {
        // A's description.
    },

    "session_B" : {
        // B's description.
    },
    ...
}
```

Each session description has to follow this format:

```
"session_A" : {
    "extends" : [],   // Array of sessions extended by this session (optional).

    "weight" : 0.0,   // Weight of this session (optional).
    "transport" : "", // Socket.IO transport used by this session (optional).

    "metadata" : {},  // Session-wide metadata (optional).
    "do" : []         // Array of actions to be performed by the user (optional).
}
```

The meaning of each of the fields is as follows:

- `extends` - an array of session names that this session extends (more about session inheritance can be found here). It is **optional** and omitting it means that this session does not extend any other sessions.
- `weight` - a real number determining how often simulated users will choose this session over other sessions (more on session selection can be found here); it is completely relative and depends on the total weight of a subset of sessions considered at one point (for examples at a certain Flood phase's startup). It is **optional** and defaults to **0.0**.
- `transport` - a string naming a Socket.IO compatible transport protocol. It should be either of `websocket` or `xhr-polling`, but in general it is **optional** and defaults to the empty string.

- `metadata` - a JSON object defining some session-wide metadata (more on metadata later). It is **optional**.
- `do` - an array of actions to be performed by the users following this session (more on actions & event handlers can be found here; a list of all available actions can be found in the next section). It is **optional** and defaults to the empty array.

3.5 User actions

Actions are performed by the simulated users after their initialization and whenever an event triggers an event handler (for example, a Socket.IO message is received or a timer is due). Actions **ordering does matter** as some actions change the state of the simulated users.
Actions are represented as short JSON arrays consisting of an `action_ID` and a JSON object listing actions `arguments`:

```
["action_ID", {
    "argument_1" : "value_1", // Argument ordering does not matter.
    "argument_2" : "value_2",
    ...
}]
```

For convenience, some actions define a shorter forms that mean exactly the same, for example:

```
["action_ID", "value_1", "value_2"] // Mind the arguments ordering.
```

The following list lists available actions, describes their effects and arguments, and gives an example invocation in both full and short forms:

- `inc` - increments a named counter either by 1 or by `Value`. Example usage:

```
["inc", "counter_name"]
["inc", "counter_name", Value]
["inc", {
    "name" : "counter_name",
    "value" : Value
}]
```

- `dec` - decrements a named counter either by 1 or by `Value`. Example usage:

```
["dec", "counter_name"]
["dec", "counter_name", Value]
["dec", {
    "name" : "counter_name",
    "value" Value
}]
```

- `set` - sets a named counter to a given `Value`. Example usage:

```
["set", "counter_name", Value]
["set", {
    "name" : "counter_name",
    "value" : Value
}]
```

- `start_timer` - starts a named timer timeouting in `Timeout` milliseconds. Example usage:

```
["start_timer", "timer_name", Timeout]
["start_timer", {
    "name" : "timer_name",
    "time" : Timeout
}]
```

- `stop_timer` - stops a named timer preventing it from timing out and triggering an event dispatch. Example usage:

```
["stop_timer", "timer_name"]
["stop_timer", {
    "name" : "timer_name"
}]
```

- `restart_timer` - restarts a named timer. Essentially, performs `stop_timer` and `start_timer` is quick succession. Example usage:

```
["restart_timer", "timer_name", Timeout]
["restart_timer", {
    "name" : "timer_name",
    "time" : Timeout
}]
```

- `timed` - executes a set of actions while timing their execution time which it then stores is a named counter. Results in whatever the actions result in. Example usage:

```
["timed", {
    "name" : "counter_name",
    "do" : [
        Action,
        ...
    ]
}]
```

- `on_timeout` - adds several timeout handlers to the simulated users state. If a given timeout handler already exists, new actions are appended **after** the existing ones, meaning they will be executed after the existing actions. Example usage:

```
["on_timeout", {
    "timer_name_1" : [
        Action,
        ...
    ],
    ...
}]
```

- `on_event` - adds several event handlers to the simulated users state. If a given event handler already exists, new actions are appended **after** the existing ones, meaning they will be executed after the existing actions. Example usage:

```
["on_event", {
    "event_1" : [
        Action,
            ...
    ],
    ...
}]
```

- `on_socketio` - adds several messages handlers to the simulated users state. If a given message handler already exists, new actions are appended **after** the existing ones, meaning they will be executed after the existing actions. Example usage:

```
["on_socketio", {
    "opcode_1" : [
        Action,
            ...
    ],
    ...
}]
```

- `emit_event` - emits `Event` with `Args` as a Socket.IO message with the event opcode. Example usage:

```
["emit_event", {
    "name" : Event,
    "args" : Args
}]
```

- `emit_socketio` - emits a Socket.IO message to the given `Endpoint` with the given `Opcode` and `Payload`. Example usage:

```
["emit_socketio", {
    "opcode" : Opcode,
    "endpoint" : Endpoint,
    "data" : Payload
}]
```

- `emit_http` - emits a synchronous HTTP request with a given `Method`, `Body`, `Headers` and `Timeout` to a given `Url`. Afterwards, executes actions defined in `on_reply` or `on_error` when the requests succeeded or failed respectively. Example usage:

```
["emit_http", {
    "url" : Url,
    "method" : Method,
    "body" : Body,
    "headers" : Headers,
    "timeout" : Timeout,

    "on_reply" : [
        Action,
        ...
    ],

    "on_error" : [
        Action,
        ...
    ]
}]
```

- `match` - performs either a JSON-based or RegExp-based pattern-matching operation on `Subject`. RegExp-based matching takes precedence over JSON-based matching. The results are stored in the simulated users metadata under `Name_#` (where # is the index of the match) for RegExp-based matching or under respective `$names` for JSON-based matching. Afterwards, executes actions defined in either `on_match` or `on_nomatch` when the matching succeeds or fails respectively. Example usage:

```
["match", {
    "name" : Name,
    "subject" : Subject,
    "re" : "regexp",

    "on_match" : [
        Action,
        ...
    ],

    "on_nomatch" : [
        Action,
        ...
    ]
}]

["match", {
    "subject" : Subject,
    "json" : {
        "field_1" : "$value_1",
```

```
        "field_2" : "$value_2",
        ...
    },

    "on_match" : [
        Action,
        ...
    ],

    "on_nomatch" : [
        Action,
        ...
    ]
}]
```

- `case` - performs a value case dispatch on a given `Value` selecting a matching `Branch` and executing its respective actions. Example usage:

```
["case", Value, {
    Branch : [
        Action,
        ...
    ],
    ...
}]
["case", {
    "condition" : Value,
    "branches" : {
        Branch : [
            Action,
            ...
        ],
        ...
    }
}]
```

- `def` - adds new metadata to the simulated users state. Example usage:

```
["def", {
    "key_1" : "value_1",
    "key_2" : "value_2",
    ...
}]
```

- `terminate` - immediately stops actions execution and terminates the simulated user with termination reason set to `Reason`. Disconnects him from the server and terminates his process. Example usage:

```
["terminate", Reason]
["terminate", {
    "reason" : Reason
}]
```

- `log` - prints a log line to the console formatting it with the `Format` and `Values`. The `Format` format is the same as Erlangs `io:format/2` (why yes, I did lie about the "no Erlang required" thing, deal with it). Example usage:

```
["log", Format, Values]
["log", {
    "format" : Format,
    "values" : Values
}]
```

- `!log` - a convenience action that allows easy `log` toggling; does nothing. Example usage:

```
["!log", Format, Values]
["!log", {
    "format" : Format,
    "values" : Values
}]
```

3.6 Metadata

Flood provides a per-user key-value store that can be accessed later by the simulated users. Various parts of a Flood scenario may define arbitrary key-value pairs in the `metadata` field. For example:

```
"metadata" : {
    "foo" : "bar",
    "bar" : [1, 2, 3],
    ...
}
```

Metadada defined in different sections has different scope. The `server` metadata is accessible by all the users. The `phase` metadata is accessible by the users spawned in that particular phase and `session` metadata is accessible by all the users following that metadata.
Metadata is **not shared** between users, instead every user accesses a unique copy. That means that the metadada can be freely modified added and removed during simulated users execution. This is the so-called *run-time metadata*.
Metadada from different sections **can and will shadow** metadada from other sections, the order is as follows (accessed from left to right):

`run-time metadata >> session metadata >> phase metadata >> server metadata`

Metadata can be accessed freely using *JSON $ubstitutions*:

```
["emit_event", {
    "name" : "$foo", // $foo --> "bar"
    "args" : "$bar"  // $bar --> [1, 2, 3]
}]
```

In general, JSON $ubstitutions can be used anywhere in the value position with the exception of **arrays of actions**, which are not substituted because they may contain their own $ubstitutions:

```
"do" : [
    "$some_action",                  // Not substituted.
    ["start_timer", "$timer", 1000] // Will be substituted when start_timer is executed.
]
```

There is some metadata that is added to the client state by default. Most of these correspond directly to the setup of different scenario sections:

- `server.host` - the server host,
- `server.port` - the server port,
- `server.endpoint` - the server endpointt,
- `server.url` - the server URL (host:port/endpoint),
- `server.sid` - the Socket.IO session ID received from the server,
- `server.heartbeat_timeout` - the Socket.IO heartbeat timeout received from the server,
- `server.reconnect_timeout` - the Socket.IO reconnect timeout received from the server,
- `server.available_transports` - the Socket.IO transports supported by the server,
- `phase.name` - the name of the *phase* the user was spawned in,
- `phase.users` - the number of users spawned in this *phase*,
- `phase.user_sessions` - the user sessions used in this *phase*,

- `phase.start_time` - the start time of this *phase*,
- `phase.spawn_duration` - the user spawn duration of this *phase*,
- `phase.test_interval` - the goal check interval of this *phase*,
- `phase.timeout` - the timeout time of this *phase*,
- `phase.goal` - the goal of this *phase*,
- `session.name` - the name of the *session* the user is following,
- `session.base_sessions` - the array of sessions extended by this *session*,
- `session.transport` - the Socket.IO transport used by this *session*,
- `session.weight` - the weight of this *session*.

3.7 Example scenarios

3.7.1 Session inheritance

This example shows session inheritance usage (more on this here). Full Flood scenario:

```
{
    "server" : {
        "host" : "localhost",
        "port" : 8080,
        "endpoint" : "/socket.io/1/"
    },

    "phases" : {
        "phase_I" : {
            "users" : 1,
            "user_sessions" : ["e"],

            "start_time" : 1000,
            "spawn_duration" : 1000,

            "timeout" : 3000
        }
    },

    "sessions" : {
        "a" : {
            "do" : [["log", "In A!"]]
        },

        "b" : {
            "extends" : ["a"],
            "do" : [["log", "In B!"]]
        },

        "c" : {
            "extends" : ["a"],
            "do" : [["log", "In C!"]]
        },

        "d" : {
            "extends" : ["b", "c"],
            "do" : [["log", "In D!"]]
        },

        "e" : {
```

```
            "weight" : 1.0,
            "transport" : "websocket",

            "extends" : ["d", "c", "b"],
            "do" : [["log", "In E!"]]
        }
    }
}
```

Sessions are composed retaining their topological ordering what ensures *sane* execution:

- session `e` extends `d`, `c` and `b` and requires them to run first **in order**,
- session `d` extends `b` and `c`,
- session `e` ensures that `b` and `c` *will* run, so `d` doesn't need to run `b` nor `c`,
- sessions `b` and `c` extend `a`,
- since session `d` requires both `b` and `c` to run and since `e` ensures that `b` and `c` *will* run, `d` only requires `a` to run first.

Flood output:

```
10:34:01.684 [notice] Running test examples/1.json
10:34:01.712 [notice] Scheduling Flood phase phase_I: 1 users every 1000 msecs (1 max)
                      starting at 1000 ms.
10:34:01.712 [notice] Scheduling Flood phase phase_I test at 3000 ms.
10:34:02.729 [notice] In A!
10:34:02.729 [notice] In D!
10:34:02.729 [notice] In C!
10:34:02.729 [notice] In B!
10:34:02.729 [notice] In E!
10:34:04.722 [notice] Flood phase phase_I reached its goal!
```

3.7.2 Ping-Pong

This example is a little more involved, it spawns 1000 users that ping a test server and measure the response time. It shows timers & counters usage (more on timers & counters here). Full Flood scenario:

```
{
    "server" : {
        "host" : "localhost",
        "port" : 8080,
        "endpoint" : "/socket.io/1/"
    },

    "phases" : {
        "pingers" : {
            "users" : 1000,
            "user_sessions" : ["pinger"],

            "start_time" : 100,
            "spawn_duration" : 100,

            "test_interval" : 100,
            "timeout" : 10000,

            "goal" : {
                "type" : "object",
                "properties" : {
                    "counters" : {
                        "type" : "object",
                        "properties" : {
```

```
                            "received" : {
                                "type" : "integer",
                                "minimum" : 1000,
                                "required" : true
                            },
                            "sent" : {
                                "type" : "integer",
                                "minimum" : 1000,
                                "required" : true
                            }
                        }
                    },
                    "timers" : {
                        "type" : "object"
                    }
                }
            },

            "metadata" : {
                "ping_timeout" : 1000
            }
        }
    },

    "sessions" : {
        "pinger" : {
            "transport" : "websocket",
            "weight" : 0.8,

            "do" : [
                ["on_socketio", {
                    "1" : [
                        ["log", "Ping ~s!", ["$server.sid"]],
                        ["emit_event", {
                            "name" : "ping",
                            "args" : ["$server.sid"]
                        }],
                        ["inc", "sent"],
                        ["start_timer", "ping", "$ping_timeout"]
                    ],

                    "5" : [
                        ["inc", "received"],
                        ["log", "Pong ~s!", ["$message.data"]],
                        ["stop_timer", "ping"]
                    ]
                }],
                ["on_timeout", {
                    "ping" : [
                        ["log", "Ping timeouted for ~s!", ["$server.sid"]]
                    ]
                }]
            ]
        }
    }
}
```

Flood output:

```
11:38:31.902 [notice] Running test examples/2.json
11:38:31.923 [notice] Scheduling Flood phase pingers: 100 users every 10 msecs (1000 max)
```

```
                    starting at 100 ms.
11:38:31.923 [notice] Scheduling Flood phase pingers test every 100 ms starting at 100 ms,
                    with timeout at 10000 ms.
11:38:32.254 [notice] Ping 912feef519889dd9866fbfaea6bfeb96218d7ce!
...
11:38:32.341 [notice] Pong {"name":"ping","args":["912feef519889dd9866fbfaea6bfeb96218d7ce"]}!
...
11:38:34.296 [notice] Flood phase pingers reached its goal!
```

Flood results show exactly how the server behaved, with minimal request processing time (with IO time) at 54 ms and maximum processing time at 523 ms (more on flood results can be found here). Additionally, various statistics are provided:


```
{
    "counters" : {
        "ws_incomming" : 2000,
        "http_outgoing" : 1000,
        "ws_outgoing" : 1000,
        "http_incomming" : 1000,
        "disconnected_users" : 0,
        "connected_users" : 1000,
        "pingers_goal_time" : 1900,
        "alive_users" : 1000,
        "all_users" : 1000,
        "terminated_users" : 0,
        "received" : 1000,
        "sent" : 1000
    },
    "timers" : {
        "ping" : {
            "min" : 54,
            "max" : 523,
            "arithmetic_mean" : 298.8575,
            "geometric_mean" : 260.985015508945,
            "harmonic_mean" : 216.292895973774,
            "median" : 347,
            "variance" : 17071.5510714286,
            "standard_deviation" : 130.658145828833,
            "skewness" : -0.387733104425692,
            "kurtosis" : -1.27787946255272,
            "percentile" : {
                "50" : 347,
                "75" : 401,
                "90" : 447,
                "95" : 463,
                "99" : 504,
                "999" : 523
            },
            "histogram" : {
                "x" : [124,184,244,304,364,454,554,654],
                "y" : [52,75,8,18,83,135,29,0]
            },
            "n" : 400
        }
    }
}
```


3.7.3 More examples

More Flood scenario examples and their results can be found in th `examples` directory of the Flood repository.

4 Flood results

4.1 Results format

- JSON structure
- Counters
- Timers
- Available statistics

4.2 Goal schemas

- JSON Schema structure
- Testing intervals
- Reaching goals
- Goal timeouts

4.3 Continuous Integration integration

- Running Flood automagically

4.4 Example results

- Sessions
- Single ping
- Continuous ping
- “3rd party” requests