Dionea - Asynchronous Distributed Debugger for Ruby and Python (Update Version for 0.8.2)

Norio Sato and Keisuke Kosuga

June 20, 2008

Majore Changes

- Supports Ruby-1.8.6.
- Supports Mongrel Server.
- Supports fastthread (Ruby-1.8.6, Mongrel).
- CPython is updated to version 2.5.2 (new patch provided) .
- Adaped to the up to date TurboGears-1.0.4.4 (see the installation part).
- Fixing bugs (see the bug tracking)

1 What is Dionea ?

The debugger which we named "Dionea" meaning "Venus fly catcher", a plant whose latin name is "Dionaea Muscipula" (see our logo in Figure 1), provides a distiributed thread-aware symbolic debugging environment[5] for distributed programs using Python[3] and Ruby[2].

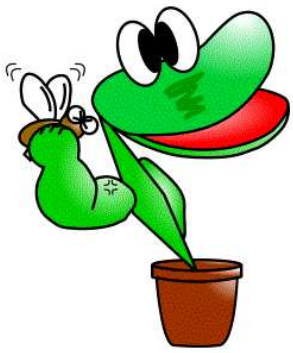


Figure 1: Logo of Dionea.

Dionea uses asynchronous one-to-many coupling of debug client (User Interface) and debug servers (target embedded debug agents). The debug client part is written in Python, while the debug sever parts are either Python (with small C code for a supplement module and a patch for CPython interpreter) or totally Ruby.

Dionea has two domain specific extensions: One is "remote stepping from caller to callee"[6]. The other is "session-aware" tracing which is an unprecedented feature[7][8].

The most important feature will be the "session-awareness". Our support by now is for Rails[1] and TurboGears[4] frameworks.

2 Project Status

Dionea basically runs on every UNIX platforms:

- Dionea by now runs on Linux, Free BSD, and OS-X, but the up todate version has not yet been verified for the latter two UNIX.
- Dionea for Windows XP/2000 has not been made available yet. For some windows PC, it woks, but for others not. One problem is PyQt whose socket sometimes does not respond. We have not fixed this problems.

Dionea depends on libraries in the standard libraries and some components by 3rd parties.

- The Dionea server parts set hooks to the libraries to raise debug events: These are in Ruby standard library (Thread; dRuby, and CGI modules), or . Python standard library (Threading), and of the web frameworks (Rails and TurboGears).
- The Dionea client part uses Qt3 and SIP for GUI interface.
- Both the client and the server parts encode the commands and responses to the YAML format to transfer them between the client and servers. To do so, Sych modules are used.

Note that not every up to date version of these libraries works with Dionea. Threfore, we provide our recommended versions in tar file, together with Dionea. (See the installation as described later.)

3 Installing Dionea

Dionea uses many other packages as below. We provide these altogether as a single tar.gz file, so that you may save time for downloading each of them separately.

- Python-2.5.x or 2.4.x
  - Python(2.5.2) [Recommend]
    - A patch of Dionea for Python-2.5.2 (dionea_py2.5.2_ceval.c.patch)
  - Python(2.5.1)
    - A patch of Dionea for Python-2.5.1 (dionea+issue1265_py2.5.1_ceval.c.patch)
  - Python(2.4.1)
    - A patch of Dionea for Python-2.4.1 (dionea_py2.4.1_ceval.c.patch)
- Ruby-1.8.6, 1.8.5 or 1.8.4
- Qt-3.x.x
- PyQt
- SIP
- python-jplib
- Syck-0.45
- uconv-0.4.11

3.1 Download need-dionea-packages-x.x.x.tar.gz

Get Download need-dionea-packages-x.x.x.tar.gz
from *http://sourceforge.net/project/showfiles.php?group_id=208017*.

```
$ tar xvzf need-dionea-packages-x.x.x.tar.gz
$ cd need-dionea-packages-x.x.x/
```

3.2 Intalling Python

Get Python-2.5.2, Python-2.5.1 *or* Python-2.4.1 from *http://python.org/*. Expand the tarball file, move to the directory, and do as follows:

```
$ patch -p1 < dionea_py2.5.2_ceval.c.patch or dionea+issue1265_py2.5.1_ceval.c.patch
or dionea_py2.4.1_ceval.c.patch
$ ./configure
$ make
$ make install
```

If Python is already installed by RPM etc, you should do the above without deleting it. The following works must be done using the newly installed Python.

3.3 Installing Ruby

Get Ruby-1.8.6, 1.8.5 or 1.8.4 from *http://www.ruby-lang.org/ja/*.

```
$ ./configure
$ make
$ make install
```

3.4 Installing Qt

Qt can be installed either by RPM or by the source code.

```
$ rpm -ivh qt-3.xx-x.rpm
$ rpm -ivh qt-devel-3.xx-x.rpm
```

3.5 Installing SIP

Get the tarball from *http://www.riverbankcomputing.co.uk/sip/index.php*. Expand this file, move to the expanded directory, and do as follows:

```
$ python configure.py
$ make
$ make install
```

3.6 Installing PyQt

Get the tarball file of PyQt from *http://www.riverbankcomputing.co.uk/pyqt/index.php*. Expand this file, move to the expanded directory, and do as follows:

```
$ python configure.py -y qt-mt
$ make
$ make install
```

3.7 Installing python-jplib

Get the file from *http://city.plala.jp/download/jplib/*.

```
$ su
$python setup.py install
```

3.8 Installing syck

Get the tarball file from *http://www.whytheluckystiff.net/syck/* Expand this file, move to the expanded directory, and do as follows:

1. Installing the body of syck

```
$ ./configure
$ make
$ make check
$ sudo make install
```

2. Installing syck-python

```
$ cd ext/python/
$ python setup.py build
$ sudo python setup.py install
```

3. Installing syck-ruby

```
$ cd ext/ruby
$ ruby install.rb config
$ ruby install.rb setup
$ sudo ruby install.rb install
```

3.9 Installing uconv

Get the tarball file from *http://www.yoshidam.net/Ruby_ja.html*. Expand this file, move to the new directory, and do:

```
$ ruby extconf.rb
$ make
$ sudo make install
```

3.10 Installing Dionea

Expand the tarball file, move to the newly created directory, and do:

- (If Using Python-2.4.1) Edit the Makefile.
  -I/usr/local/include/python2.5 -I/usr/include/python2.5 ⇒
  -I/usr/local/include/python2.4 -I/usr/include/python2.4

```
$ make
```

4 How to Connect Dionea to Debugees ?

Remote debugging, i.e., connecting and launching remote processes, is essential to debug communicating processes. As shown in Figure 2, we hereafter refer to the debugged process as "debuggee", the debugger as "debug client", and the debug agent in the *debuggee* process as "debug server".

The *debuggee* is run by the *debug server* as follows: The main thread of the *debug server* loads (simply done by *require* in Ruby and *import* in Python) and attaches hooks (done by the assignments to the methods, in dynamic languages such as Ruby and Python) at the thread libraries to generate debug events such as thread

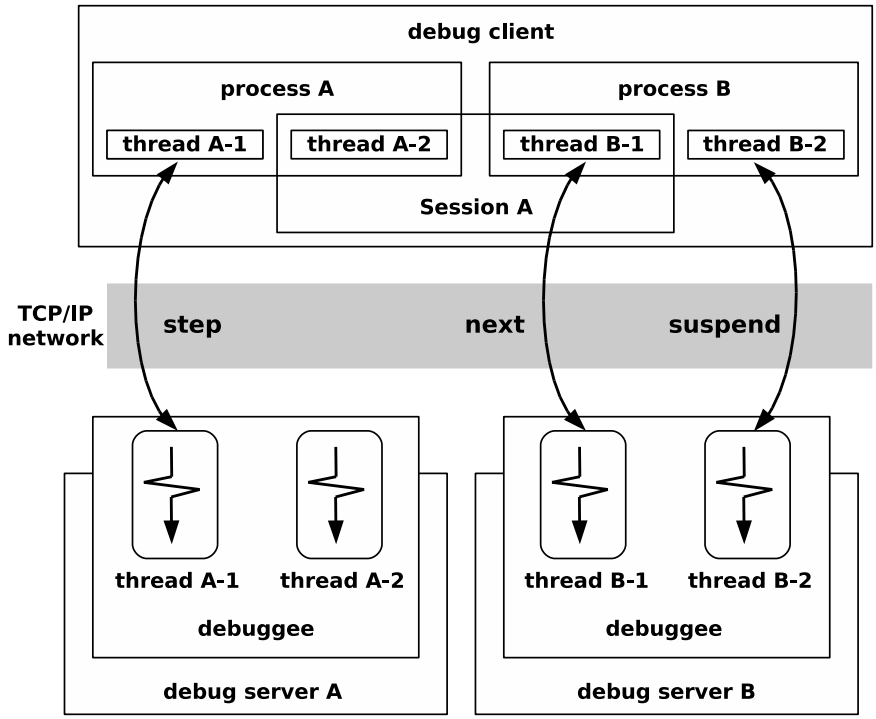


Figure 2: Dionea architecture.

creation, termination, etc., that are not supported by a debug API callback by the interpreter. Then, the main thread creates a dedicated (hidden) thread, called a "command listener", to receive debug commands and to send back responses. It then executes the specified *debuggee* script with its context as main. Thus, the *debuggee* thread(s) and the *command listener* run concurrently in the same process. The single process web server used for testing or small to mediam size sites such as WEBrick and CherryPy are the *debuggee* script itself.

With production web servers, the *debugees* are the backend processes. For example, their initial code is "dispatch.fcgi" driven by FastCGI processes , which are created by the frontend process, i.e., Apache. The user must add a single line code to load and run the *debug server* (see 7.2).

There are two ways to make this happen as follows:

- **Manual connection mode:** If the user wants to connect the *debug client* to an already runing *debuggee* process, this is appropriate. Edit the file "config/config.yml" as follows:

```
auto_connect:
 enable:  false
 client_host:  < blank >
 client_port:  < blank >
```

  The user can let the *debug client* "connect", "disconnect", and "re-connect" to a running *debug server*, or to "launch" it.

- **Automatic connection mode:** If the user wants the *debug client* automatically to a newly created *debugee* process, such as the backend processes of web servers, this is appropriate. Edit the file "config/config.yml" as follows:

```
auto_connect:
 enable:  true
```

```
client_host:  localhost -- where the debug client works
client_port:  4000 -- connection port number
```

From the connection dialog window as shown in Figure 3, choose "auto" radio button, you let the *debug server* inform its hostname and port number to the *debug client*. The hostname and the port number of the latter is specified by a configuration file for Dionea. The user transmits the port (e.g., 4000) to the *debug client* through a dialog window, which makes the connection from the *debug client* to the *debug server* happen automatically.

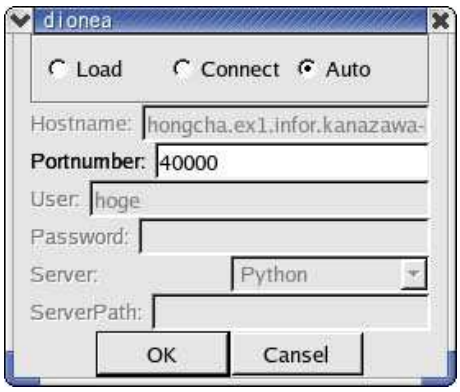


Figure 3: Connection dialog.

5 Getting Started

5.1 Sample Programs

In */path/to/dionea/sample*, two sample web projets using Rails, two sample projects using TurboGears, and several sample scripts to test multi-threading are included:

- Depot application using Rails: *.../sample/rails/depot*
- Depot application using Rails/Ajax: *.../sample/rails/depot-ajax/*
- Depot application using TurboGears: *.../sample/tg/depot/*
- Depot application using TurboGears/Ajax: *.../sample/tg/depot-ajax/*
- The dining philosophers problem: *.../sample/python/philosophers.py, .../sample/ruby/philosophers.rb*
- Producer-consumer pattern: *.../sample/python/testP-C.py, .../sample/ruby/testP-C.rb*
- Worker-thread pattern: *.../sample/python/testWorkerThread.py*
- Thread-per-message pattern: *.../sample/python/testTperM.py, .../sample/ruby/testTperM.rb*
- Read/write lock synchronization: *.../sample/python/testRWLock.py*
- Barrier synchronization: *.../sample/python/testM-T.py*
- dRuby sample programs: *.../sample/ruby/druby/test-client.rb,test-server.rb*

When using "Apache or lighttpd" (see 7.2), these web applications must be placed to an appropriate directory which agrees with the "document root" that you set in the configuration of the web server, e.g., "httpd.conf" of Apache (see 6.3.2), "lighttp.conf" of lighthttpd (see 6.4.1). For example:

```
$ cd /path/to/dionea/sample
$ cp -r rails ~/public_html/
```

When using only WEBrick or CherryPy, this is not always necessary.

5.2 Getting Started with non-Web Sample Scripts

How to run the sample code which are not web projects are descripbed as following:

1. *Running debug client:*

   ```
   $ cd /path/to/dionea
   $ dionea &
   ```

   If the "automatic connection" is set enabled in the configration file ("path/to/dionea/config/config.yml"), you have to run the debug client in the same machine that you specified in this configuration file. Click the "connect" manu, and a dialog window is open. Then, choose the "auto" radio button. Then input the same port number that is specified in the configuration file.

2. *Running a debuggee*

   A Python debuggee is run by the Python debug server as:

   ```
   $ cd /path/to/dionea
   $ dioneas.py [-p <port number>] sample/script-name.py &
   ```

   A Ruby debuggee is run by the Ruby debug server as:

   ```
   $ cd /path/to/dionea
   $ dioneas.rb [-p <port number>] sample/script-name.rb &
   ```

3. Connecting the debug client to a debuggee (see 4)

   By default, the connection is "manual". In this case, clock the "connect" menu in the toolbar, and a dialog window appears. Then, select the "connect" radio button, and specify the *port number* specified as above, and the name of the host where the debuggee is run.

   If you have specified the connection as to be "automatic" in the configuration file ("config.yml"), the connection happens automatically at each time a new debuggee run (by the debug server).

How to run Dionea with the web projects is described in the following sections.

6 Ground Works on Rails

6.1 Installing MySQL

The web application samples (depot) use databases, e.g., MySQL. Download by RPM or by the source code. Download site: *http://www.mysql.com/*

```
$ /usr/sbin/groupadd mysql
$ /usr/sbin/useradd -g mysql mysql
$ ./configure [--with-charset=ujis]
--with-extra-charsets=all
--with-mysqld-user=mysql
--with-unix-socket-path=/var/lib/mysql/mysql.sock
--enable-thread-safe-client
```

```
$ make
$ make install
$ mysql_install_db --user=mysql
$ chown -R root /usr/local/mysql
$ chgrp -R mysql /usr/local/mysql
$ mysqladmin -u root password <root passwd>
$ mysqld_safe [--old-passwords] &
```

6.2 Installing Rails (1.2.6)

1. Install RubyGems-0.9.4 (*http://rubyforge.org/frs/?group_id=126*) if missing. Download, expand the file and move to the created directory and do:

   ```
   $ ruby setup.rb
   ```

2. Install Rails as follows:

   ```
   $ gem install -r rails --version '= 1.2.6'
   $ gem install mysql
   ```

6.3 Using Apach, FastCGI Server

6.3.1 Installing Apache

Get the tarball file from *http://www.apache.jp/*.

```
$ ./configure -enable-so [--with-mpm=worker]
$ make
$ sudo make install
```

Installing mod_rewrite (DSO).

```
$ cd /path/to/apache-source
$ /usr/local/apache2/bin/apxs -c modules/mappers/mod_rewrite.c
$ /usr/local/apache2/bin/apxs -i -a -n rewrite modules/mappers/mod_rewrite.la
```

6.3.2 Installing FastCGI

1. Download the file from *http://www.fastcgi.com/*.

   ```
   $ ./configure
   $ make
   $ sudo make install
   ```

   Add "/usr/local/lib" to "/etc/ld.so.conf".

2. Install the *ruby-fastcgi* by gems:

   $ gem install fcgi

3. Install mod_fcgi (http://www.fastcgi.com/)

```
$ cp Makefile.AP2 Makefile
$ make top_dir=/usr/local/apache2
$ sudo make install
```

4. Setup “httpd.conf” as follows:

```
LoadModule rewrite_module modules/mod_rewrite.so
LoadModule fastcgi_module modules/mod_fastcgi.so

<IfModule mod_fastcgi.c>
 FastCgiConfig -idle-timeout 30 [-maxClassProcesses n]
               -initial-env RAILS_ENV=development
 FastCgiIpcDir /var/run/fastcgi
</IfModule>
<VirtualHost *:8080>
 DocumentRoot /home/hoge/public_html/rails/depot/public
 errorLog /home/hoge/public_html/rails/depot/log/server.log
 AddDefaultCharset UTF-8
 <Directory /home/hoge/public_html/rails/depot/public>
  AddHandler fastcgi-script .fcgi
  AddHandler cgi-script .cgi
  Options +FollowsymLinks +ExecCGI
  RewriteEngine On
  RewriteRule ^$ index.html [QSA]
  RewriteRule ^([^.]+)$ $1.html [QSA]   RewriteCond %REQUEST_FILENAME !-f
  RewriteRule ^(.*)$ dispatch.fcgi [QSA,L]
 </Directory>
 ErrorDocument 500 ‘‘&lt;h2&gt;Application errorh&lt;/h2&gt;Rails application
failed to start properly’’
</VirtualHost>
```

6.4 Using lighttpd, FastCGI Server

6.4.1 Installing lighttpd

1. Install lighttp Download site: *http://www.lighttpd.net/*. Do as follows:

```
$ ./configure
$ make
$ sudo make install
```

2. Start the server as follows:

```
$ /usr/local/sbin/lighttpd -f /path/to/lighttpd.conf
```

3. Set the configuration file (lighttpd.conf) as follows:

```
server.port=3000
server.modules=("mod_rewrite","mod_fastcgi")
url.rewrite=("^/$"=>"index.html","^([^.]+)$"=>"$1.html")
server.error-handler-404="/dispatch.fcgi"
server.document-root="/home/hoge/public_html/rails/depot/public"
server.errorlog="/home/hoge/public_html/rails/depot/log/server.log"

fastcgi.server=(".fcgi"=>
 ("localhost"=>
```

```
  (
   "min-procs"=>1,
   "max-procs"=>5,
   "socket"=>"/tmp/application.fcgi.socket",
   "bin-path"=>"/home/hoge/public_html/rails/depot/public/dispatch.fcgi",
   "bin-environment"=>("RAILS_ROOT"=>"/home/hoge/public_html/rails/depot",
      "RAILS_ENV"=>"development")
  )
 )
)

mimetype.assign=( ".css"=>"text/css", ".html"=>"text/html", ".htm"=>"text/html",
".js"=>"text/javascript", ".png"=>"image/png", ".jpeg"=>"image/jpeg", ".jpg"=>"image/jpeg"
)
```

6.4.2 Installing FastCGI

See 6.3.2.

6.5 Using Mongrel Server

6.5.1 Installing Mongrel

Install Mongrel (*http://mongrel.rubyforge.org/*) as follows:

```
$ gem install mongrel
```

6.6 Using Apache, mod_proxy, mod_proxy_balancer, mongrel, mongrel_cluster Server

6.6.1 Installing Apache, mod_proxy, mod_proxy_balancer

1. Installing Apache.
   See 6.3.1.

2. Install mod_proxy, mod_proxy_balancer

```
$ cd /path/to/apache-source
$ /usr/local/apache2/bin/apxs -c modules/proxy/mod_proxy.c modules/proxy/proxy_util.c
$ /usr/local/apache2/bin/apxs -i -a -n proxy modules/proxy/mod_proxy.la

$ /usr/local/apache2/bin/apxs -c modules/proxy/mod_proxy_balancer.c
                                modules/proxy/proxy_util.c
$ /usr/local/apache2/bin/apxs -i -a -n proxy_balancer modules/proxy/mod_proxy_balancer.la

$ /usr/local/apache2/bin/apxs -c modules/proxy/mod_proxy_http.c modules/proxy/proxy_util.c
$ /usr/local/apache2/bin/apxs -i -a -n proxy_http modules/proxy/mod_proxy_http.la
```

3. Set the configuration file (httpd.conf) as follows:

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule proxy_http_module modules/mod_proxy_http.so

<VirtualHost *:8080>
```

```
DocumentRoot /path/to/dionea/sample/rails/depot/public
AddDefaultCharset UTF-8
ProxyRequests Off

<Directory /path/to/dionea/sample/rails/depot/public>
 Options FollowSymLinks
 AllowOverride None
 Order allow,deny
 Allow from all
</Directory>

ProxyPass /images !
ProxyPass /stylesheets !
ProxyPass /javascripts !
ProxyPass /favicon.ico !

ProxyPass / balancer://depot/
ProxyPassReverse / balancer://depot/

<Proxy balancer://depot/>
 BalancerMember http://127.0.0.1:3000 loadfactor=10
 BalancerMember http://127.0.0.1:3001 loadfactor=10
 BalancerMember http://127.0.0.1:3002 loadfactor=10
</Proxy>
</VirtualHost>
```

4. Start the server as follows:

```
$ /usr/local/apache2/bin/httpd -k start
```

6.6.2 Installing Mongrel

See 6.5.1.

6.6.3 Installing Mongrel_cluster

1. Install Mongrel_cluster
   Install Mongrel_cluster (*http://mongrel.rubyforge.org/wiki/MongrelCluster*) as follows:

```
$ gem install mongrel_cluster
```

2. Set the configuration file (/path/to/rails-app/config/mongrel_cluster.yml) as follows:

```
$ cd /path/to/rails-app
$ mongrel_rails cluster::configure -e development -p 3000 -N 3
```

3. Start the server as follows:

```
$ cd /path/to/rails-app
$ mongrel_rails cluster::start

Stop the server :
$ mongrel_rails cluster::stop
```

6.7 Setup the Rails Depot Application

1. **Create the Database:**
   Username is hoge. Hostname is localhost.

   ```
   $ mysql -u root -p
   Enter passwd *****
   mysql> CREATE DATABASE depot_development;
   mysql> GRANT ALL PRIVILEGES ON depot_development.* TO hoge@localhost IDENTIFIED
   BY 'password';
   mysql> exit
   ```

2. **Create the Tables:**

   ```
   $ mysql -u hoge -p depot_development < /path/to/dionea/sample/rails/db/create.sql
   ```

3. **Create the DB datas:**

   ```
   $ mysql -u hoge -p depot_development < /path/to/dionea/sample/rails/db/insert.sql
   ```

4. **Configure the application:** Edit the configuration file(config/database.yml).

   ```
   development:
     adapter:  mysql
     database:  depot_development
     username:  hoge
     password:  password
     socket:  /var/lib/mysql/mysql.sock
   ```

7 Getting Started: Depot of Rails

7.1 Using WEBrick Server

1. Create a new terminal, and do as follows:

   ```
   $ cd /path/to/dionea
   $ dionea &
   ```

2. **Start the debug server and connect:** You can connec the debug server to WEBrick both by the manual and automatic connection modes.

   - Using the manual connection mode:
     (a) Edit the Dionea configuration file "config/config.yml" (see 4).
     (b) Crete a new terminal, and do as follows:
         $ cd /path/to/dionea/sample/rails/depot/
         $ /path/to/dionea/dioneas.rb [-p 1000] ./script/server
     (c) Click the connect manu in the toolbar, and the connection dialog appears. Select the connect radio button, and input the server host name (e.g., localhost) and the port number (1000).
   - Using the automatic connection mode:
     (a) Edit the Dionea configuration file "config/config.yml"(see 4):

(b) Click the connect menu in the toolbar, and the connection dialog window appears. Select the auto radio button, and input the port number 4000.

(c) Create a new terminal, and do as follows:

```
$ cd /path/to/dionea/sample/rails/depot/
$ /path/to/dionea/dioneas.rb [-p 1000] ./script/server
```

(d) Then WEBrick starts and you can observ the main thread suspends. Click the continue icon in the toolbar, and the WEBrick resumes.

3. **Run your browser, e.g., FireFox and trace** Start your browser e.g., FireFox and specify the URL as *http://localhost:3000/store*

7.2 Using (Apach or lighttpd), FastCGI Server

1. Add this code to the head of "~/public_html/rails/depot/public/dispatch.fcgi"

```
require "/path/to/dionea/dioneas"
```

2. Set dionea configuration file (config/config.yml) as:

```
auto_connect:
 enable:  true
 client_host:  localhost
 client_port:  4000
```

3. Create a new terminal, and run the debug client as follows:

```
$ cd /path/to/dionea
$ dionea &
```

4. Click the connect icon in the toolbar, and the dialog window appears. Select the auto radio button, and input the port number as 4000, which is set in the configuration file.

5. Start your browser e.g. FireFox, and specify the URL:

   - *http://localhost:8080/store*, to use Apache,
   - *http://localhost:3000/store*, to use lighttpd

7.3 Using Mongrel Server

1. Set dionea configuration file (config/config.yml) as:

```
auto_connect:
 enable:  true
 client_host:  localhost
 client_port:  4000
```

2. Create a new terminal, and run the debug client as follows:

```
$ cd /path/to/dionea
$ dionea &
```

3. Click the connect icon in the toolbar, and the dialog window appears. Select the auto radio button, and input the port number as 4000, which is set in the configuration file.

4. Start dionea_mongrel_rails server. And connect dionea server(Auto).

```
$ cd /path/to/dionea/sample/rails/depot/
$ /path/to/dionea/dioneas_mongrel_rails start
```

5. Start your browser e.g. FireFox, and specify the URL:
   http://localhost:3000/store

7.4 Using Apache, mod_proxy, mod_proxy_balancer, mongrel, mongrel_cluster Server

1. Set dionea configuration file (config/config.yml) as:

```
auto_connect:
 enable:  true
 client_host:  localhost
 client_port:  4000
```

2. Create a new terminal, and run the debug client as follows:

```
$ cd /path/to/dionea
$ dionea &
```

3. Click the connect icon in the toolbar, and the dialog window appears. Select the auto radio button, and input the port number as 4000, which is set in the configuration file.

4. Start dionea_mongrel_rails server. And connect dionea server(Auto).

```
$ cd /path/to/dionea/sample/rails/depot/
$ /path/to/dionea/dioneas_mongrel_rails cluster::start

Stop the server:
$ /path/to/dionea/dioneas_mongrel_rails cluster::stop
```

5. Start your browser e.g. FireFox, and specify the URL:
http://localhost:8080/store

8 Tracing the Depot (of Rails/TurboGears)

We provide users with the following "session-aware" features:

(Feature A) Automatic acquisition of session ID and snapshot of threads grouped by session ID(s): An asynchronous snapshot of threads grouped by session ID(s) is provided. Using this, we select the session of interest.

(Feature B) For a specified session, breakpoint setting at an action, and its enabling/disabling: Often in web frameworks, the application is implemented using the Model-View-Controller (MVC) pattern. In such cases, the appropriate default breakpoints are the instance methods of controllers that are the effective starting point of *web scripts*, which are called *actions* in Rails.

We provide users with these features by an interface that is independent of the running platforms, i.e., web servers, and transparent to the users. Figure 4 shows an example of using the proposed features (A) and (B), which is a snapshot. The upper half shows the browser display, while the lower half is the debugger view, and the arrows indicate transitions.

This application consists of "purchaser session(s)" and an "administrator session". We first debug a "purchaser session". Figure 5 shows a snapshot of sessions and threads, where we observe three threads and processes involved in two sessions. Using feature (A), we can select the purchaser session.

As shown in Figure 6, enabling and disabling of feaure (B) is a one-touch operation.

1. When the application breaks, click the continue icon in the toolbar.

2. On the Dionea GUI, when the application breaks, you can specify "disturb-action mode" to the current session. Click the disturb mode pulldown menu as shown in Figure 6, and set the distrub-action alternative to on. Otherwise, if you want to trace other sessions, switch the session do as follows: In the session tree tub in the session view, clock a thread indication of this session.

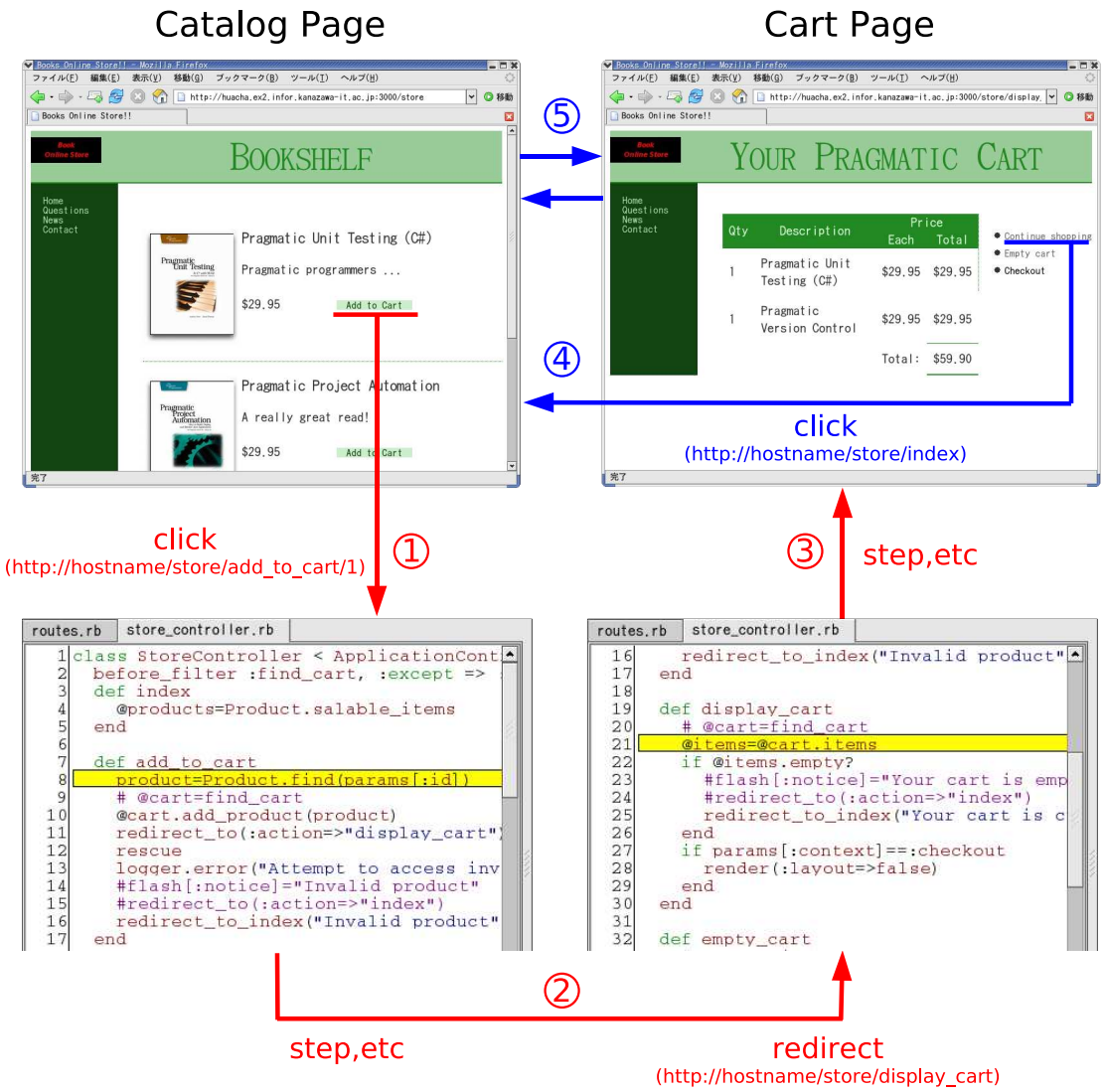


Figure 4: Automatic breaks at controller actions.

3. Click the continue icon, then as shown in Figure 4, the catalog page will be rendered on your browser, then click the tag "add to cart". On the Dionea GUI, the debuggee will break at "StoreController#add_to_cart". Then try, as follows:

   (a) Continue debugging using steps etc., which causes a server thread by redirection to break at "StoreController#display_cart".

   (b) Continue debugging further using steps etc., the cart page is displayed on the browser.

   (c) Disable disturb-action mode, click the "continue shopping" tag on the cart page, which causes a server thread to execute "StoreController#index" without any break.

   (d) Repeat these steps, i.e., (a) to (d), without enabling disturb-action mode, we see the catalog and the cart pages alternate without any break.

Next, we may want to check if the order record is produced after the purchaser clicked the "checkout" tag. Using another browser, we select the "administrator session" using feature (A). Initially we may disable feature (B) for this session. When a problem is found, we should re-enable this feature for debugging. While

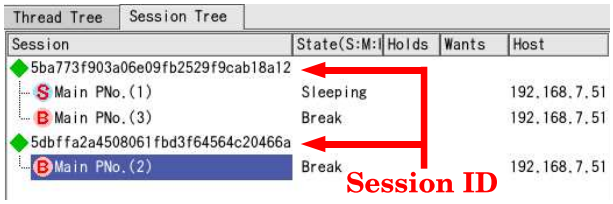


Figure 5: Displaying threads grouped by session ID.

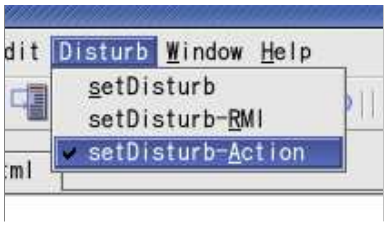


Figure 6: One-touch enabling and disabling automatic breakpoint setting.

feature (B) is disabled for the purchaser session, the debugging of the administrator session has no direct influence on the operations of steps 1 to 4. By enabling feature (B) for both sessions, we can interleave the debugging of both sessions.

9 Ground Works on TurboGears

9.1 Installing TurboGears

1. **Installing setuptools:** Download "ez_setup.py" from *http://peak.telecommunity.com/dist/ez_setup.py*, and do:

   ```
   $ ./ez_setup.py setuptools
   ```

2. **Installing TurboGears:**

   $ easy_install TurboGears==1.0.4.4
   $ easy_install SQLObject
   $ easy_install Scriptaculous
   $ easy_install MySQL-python

10 Setup the TurboGears Application Depot

The datebase creation is the same as for the Rails Depot. See Section 6.7. If already done, you can skip it.

11 Getting Started with Depot TurboGears

1. **Create the Database:**
   Username is hoge. Hostname is localhost.

   $ mysql -u root -p
   ```
   Enter passwd *****
   mysql> CREATE DATABASE depot_development_tg;
   mysql> GRANT ALL PRIVILEGES ON depot_development_tg.* TO hoge@localhost IDENTIFIED
   BY 'password';
   ```

```
mysql> exit
```

2. **Configure the application:**
   Edit the configuration file(dev.cfg).

   ```
   sqlobject.dburi="mysql://hoge:password@localhost/depot_development_tg"
   ```

3. **Create the Tables:**

   ```
   $ cd /path/to/dionea/sample/tg/depot
   $ tg-admin sql create
   ```

4. **Create the DB datas:**

   ```
   $ mysql -u hoge -p depot_development_tg < /path/to/dionea/sample/tg/db/insert.sql
   ```

5. **Run CheeryPy:** The configuration of Dionea is reusable (see 7). The way to run CherryPy is similar that to run WEBrick, as follows:

   ```
   $ cd /path/to/dionea/sample/tg/depot
   $ /path/to/dionea/dioneas.py [-p 1000] ./start-depot.py
   ```

6. **Start the debug client:**

   ```
   $cd /path/to/dionea
   $./dionea &
   ```

   Click the connect menu icon, and the connection dialog appears. If the automatic connection mode is set in the configuration file "/path/to/dionea/config/config.yml", input the connection port number, e.g.,4000.

7. Start you browser, e.g., FireFox and specify the URL:

   http://localhost:8080/store

   (Here, we assume Apache is not runnning.) Then, the application breaks. On the Dionea GUI, click the disturb-mode menu in the toolbar, set the disturb-action, and click the continue icon in the toolbar.

12 Tracing a TurboGears Application

The debug interface is exactly the same as that for the Rails application. See Section 8.

13 Limitations and Future Works

- **GUI toolkit:** The adaption to Qt 4 is out of scope for now.
- **View stepping:** Stepwise trace of the view description langugage (kid) for TurboGears is not available.
- **Automatic code update:** Automatic code update saves the operation to restart a debug session.
  - Rails has the feature to reload modified source code at *development* phase, but not at *production* phase. The internal mechanism is complicated and we have not fully understood yet.
  - "CherryPy" of TurboGears supervises modified source code and restarts the server process. Because the latter mechanism is useful for other applications besides web scripts, we have implemented it in Dionea *debug server* for Python. As a consequence, the debuggee of Python has two processes: one for the debuggee script execution, and the other for supervising code modification.

  - Problem: Dionea always displays the updated code, whether it is loaded or not. This may not be appropriate in the *production* phase.

- **Source code hierarchy:** Establishing the entire source code tree of the debuggee based upon its "load-path" at connection time takes time, particularly with Rails. Doing this on-demand basis increases efficiency.

- **Sessions and browsers:** For the user to know the browser from the session view is confusing. More information, e.g., IP address, browser name, should be added to the view.

- **Rerun:** How to "rerun a debuggee process" is under reconsideration.

  - The main thread of Dionea whose number is 1 is the main thread of the debugee process. The listener is given number 2.
  - The "Rerun" will be implemented by killing other threads, and let the main thread reload the debuggee script.
  - Problem: The threads of "sleeping state" cannot be killed except for some input.

- **Performace issue:** Performance degradation of debuggees is a drawback of Dionea. This is due to the line-by-line callback of the debug API, i.e., using tracing to detect breakpoints. The execution speed of virtual instructions is slowed down by two orders of magnitude, and threads that are i/o bound, by one order of magnitude. Thus Dionea is clearly not recommended for the *production* phase, where high performance is required.

- Due to the same reason, the initial starting time of the *debug server* is also an issue with those web frameworks that involve a large amount of library invocations. Effective improvements:

  - to skip library code,
  - to disable the callbacks by the main thread,
  - to set callbacks only for the threads that execute the sessions with *disturb-action-mode* enabled.

  The first is done and makes effect. The second proved to be effective for TurboGears. The second and the third remedy is impossible with Ruby debug API. Summing up, some efficient way to detect breakpoints is highly desirable.

14 Bugs

——-

References

[1] Dave Thomas, David Heiemeier Hansson, Leon Breed, Mike Clark and Andreas Schwarz: Agile Web Development with Rails, A Pragmatic Guide, Pragmatic Bookshelf, 2005.

[2] Dave Thomas, Chad Fowler and Andy Hunt: Programming Ruby: The Pragmatic Progrmmer's Guide, Second Editon, Pragmatic Bookshelf,2006.

[3] Mark Lutz and David Acher: learning Python, Addition Wesley, 1993.

[4] Mark Ramm, Kevin Dangoor and Gigi Sayfan: Rapid Web Applications with TurboGears, Prentice-Hall, 2006.

[5] Norio Sato, Kazuhiro Nagai, Yasushi Itoh, Masamitsu Ogura and Keisuke Kosuga: Low-intrusion Debugger for Python and Ruby Distributed Milti-thread Programs, Vol.45, No.12, pp2741-2751,IPSJ Journal (Dec. 2004)

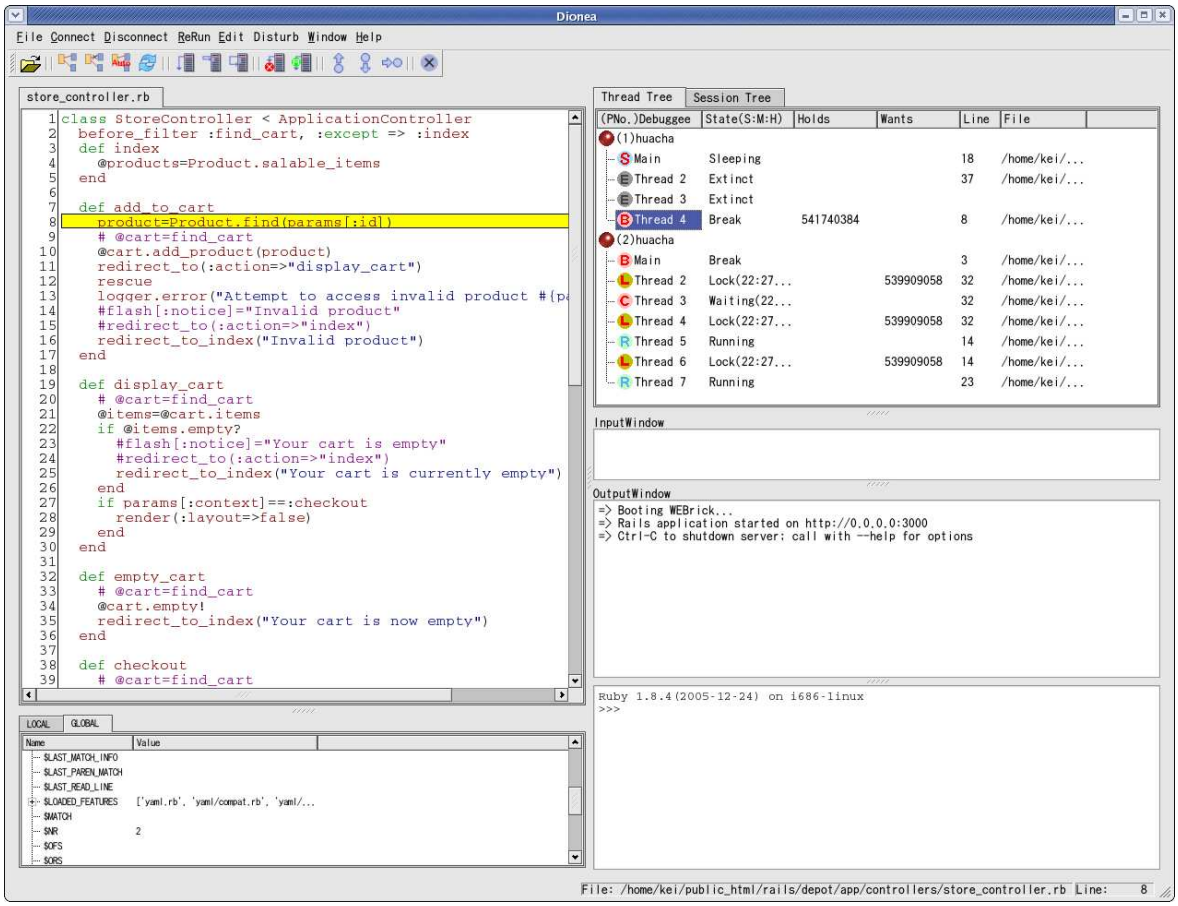


Figure 7: User interface.

[6] Keisuke Kosuga, Masamitsu Ogura and Norio Sato: Automatic Recognition and Tracing of Peer Threads by Low-intrusion Type Debugger for Python and Ruby, Vol.47, No. SIG 11(PRO 30), pp13-27, IPSJ Journal: Programming (July 2006). (Written in Japanese. Informal English version will be provided.)

[7] Keisuke Kosuga and Norio Sato: Session-Aware trace of Web Applications by One-to-Many Asynchronously Coupled Debugger, Vol.48, No. SIG 10(PRO 33), IPSJ Journal:Programming (June 2007). (Written in Japanese. Informal English version will be provided.)

[8] Norio Sato and Keisuke Kosuga: Session-aware Debugging Features for Web Applications using Ruby and Python Frameworks, Proceedings of Domain Specific Testing Automation (DoSta'07) pp13-19. Workshop of ESEC/FSE-07 (Sep. 2007), ACM digital library.

A User Interface

User interactions to control individual *debuggee* threads are independent of each other. We call them *debug sessions*. The *debug sessions* must be interleaved but should not be inter-mixed in user interface. The *debug client*, accepts one or more break or step events at the same time, and reacts for the *debug session* that the user wants. Dionea interfaces with users by means of GUI. As shown in Figure 7, the GUI consists of views and a tool bar.

- **Debuggee view (the right-hand side upper):** A tree form shows connected *debuggees* as its nodes and their threads as their leaves. Each leaf has an indicator to show the thread status: either R(running), E(Extinct), E!(Extinct by exception), B(break suspended), L(locked), C (conditional wait)

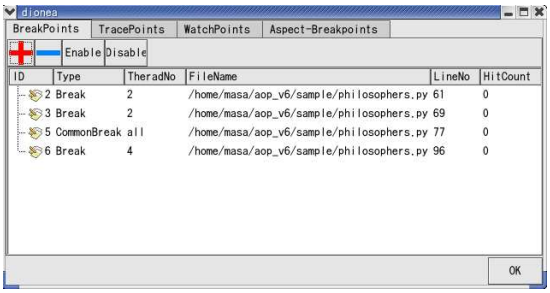


Figure 8: Breakpoints dialog.

or S (sleeping), which is followed by lock objects it is trying to hold (wants <object list>) and those it is holding (holds<object list>). Here, we add another view of threads grouped by sessions.

- **Thread context:** The following views which we call *thread context* can be switched from those for one thread to those for another by clicking a leaf:
  - **Source code view (left hand side middle):** The line of the code that the thread is about to execute or executed last before "calling native code" is highlighted.
  - **Command shell (right-hand side lower):** This is CUI interface of the debugger.
  - **Variable view (left hand side lower):** Values that are bound to the global and local variables are displayed in tree form that is independent of specific language syntax.
- **Process context:** The following views which we call *process context* are switched for the thread context switch across different process:
  - **Input and output views (right-hand side middle):** These are not used for debugging web scripts, otherwise the standard input and output of the *debuggee* are redirected here.
  - **Breakpoint dialog:** The "line type" breakpoints are set by clicks on the *source view*, while tracepoints, data breakpoints and pointcuts[1] are set using dialog windows. Opening via an icon at tool bar as shown in Figure 8, a list of break and trace points appear. The leftmost two tabs are for the "line type" break and trace points that are set by clicks on the source view. The rightmost two are data breakpoints and cutpoints, which are set using dialog windows. Figure 9 shows the dialog to set a data breakpoint.

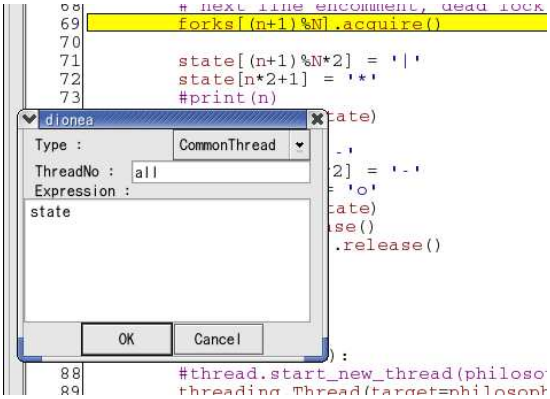


Figure 9: Tracepoints dialog.

[1] A term of Aspect Oriented programming. In this paper it means a set of breakpoints with advice code.

- **Tool bar (upper):** Pop-up views and dialog are: *source code file tree* and *break point editing dialog* which belong to a *process context*, and *connection disconnection dialog*. One touch command icons are: *continue/step/next*, *suspend*, *rerun*, *up/down*, and *quit*. Here we provide *disturb modes* setting manu.

B Commands and Icons

Dionea provides with a superset of the commands of conventional interactive debuggers, which are for:

- **getting source code files:** The source code for scripting languages always resides in the computer where the *debuggee* runs. The source code tree (click popup icon) is built at connection time using the load-path directories of the *debuggee*. The contents of source files are obtained on-demand automatically and cached. Source code requesting commands are issued when a node of the source code tree is clicked, and when needed to show the *thread context*. The source code shown in the *source code view* is always up to date, but the breakpoints are not automatically updated.

- **getting thread status:** "status <*thread number*>" is a means to know the thread status, which is implicitly issued when the *thread context* is switched to be current. "statusList" is an inquiry of the whole thread status to the *debug server*. These commands are debugger internal and therefore, the users need not to use them.

- **suspending and resuming threads**("suspend/resume <*thread number*>") "suspend <thread number>" or "resume <thread number>" causes the specified running thread to be suspended at next line, or to be resumed. The *thread number* by default is taken from the current *thread context*.

- **setting, enbling, disabling break and trace points**

  - The most low-intrusive is a "per-thread break point" that is effective only for the specified thread. The command syntax for setting this is "break *file:line* <*thread number*>". Nextly low-intrusive is a "common break point" such that any thread breaks that hits the break point (but does not suspend other threads). The command syntax is "commonBreak *file:line*". A "process break point" is *high-intrusive*, which is set by "processBreak *file:lineno*" command.
  - Any instrumentation code (given as *expression* or *string*) can be added to a break point, which we call "trace point". A trace point does not cause the thread that hits it to be suspended, unless the *expression* (*string*) causes suspension, which allows very low-intrusive investigation of thread behavior. Trace points are either per-thread or common to threads.
  - A "temporary break point" is disabled once hit. We realized this by attaching a hit counter to a break point, where "0" means permanent.

  The data break/trace points, aspect-oriented break/trace points are set by the *breakpoint dialog*, which is popped up by the toolbar icon.

  Break points can be "enabled" or "disabled" or "deleted", by clicking the *souce code view* or by the *breakpoint dialog*.

- **stepping** "step ", "next", "return", and "continue"commands can be input by icons. Their functinalities are the same as those of traditional debuggers, except that a *thread number* is attached, that is by default taken from the current *thread context*

  The semantics of the *step* command is extended, so that it may cause step into and return from remote methods in a location transparent way.

- **stack investigation:** "up", "down" and "where" commands are the same as those of traditional debuggers, except for the *thread number* attached, which is by default taken from the current *thread context*. *up* and *down* can be done by icons.

- **value evaluation and setting:** "print *expression*", "display *expression*" and "exec *string*" request the evaluation result of the specified *expression* or *string* with the context (active stack frame and its global dictionary) of the specified thread, which is identified by the current *thread context*. Setting a value to a variable is done by using these commands.
- **high-intrusive operations:** The implementation pending.
- **One-touch rerun:** The *command listener* forced to terminate all the threads according to the manner of Ruby and Python, using "asynchronous exception" mechanism[2].

  All the "debug objects" associated with the threads are deleted, and the debugee script files are reloaded and executed from scratch. On the *debug client* side, thread intrinsic breakpoints and source code is deleted.

Icons enable automatic command parameter generation in most cases.

C Basic Features of Dionea

C.1 Concept of Asynchronous Thread-awareness

C.1.1 Low-intrusive Operations

Dionea catches one more more debuggee processes at the same time, has an asynchronous snapshot on threads inside of these processes. Dionea enables us to break, step, resume, etc., individual threads with other threads directly unaffected. We call this "low-intrusive"environment, for the asynchronous world of Python and Ruby. The "low-intrusive" environment enables us to check the behavior of individual threads, by breaking, stepping, tracing, resuming and suspending, etc., while the execution of other threads remain unaffected. This contrasts with the "stop the world" or "high-intrusive" environment that traditional debuggers provide.

The low-intrusive operations enable us to debug a process without stopping its interactions with the outside world so that the debugged process may continue with its working load by receiving input. For example, it is useful for checking producer consumer threads. It enables us to cause scheduling perturbations at debug time, which offers many chances to detect synchronization bugs. For example, it is useful for checking resource sharing among threads, and more complex synchronizations such as read-write locks and barriers. It enables us to input debug commands while the debugged process is running. None of these is possible in the "high-intrusive" environment.

C.1.2 Asynchronous Snapshot of Threads

Dionea has an asynchronous snapshot of threads, which is updated by the reports coming from running threads that hit hooks in the debugged processes that we call *debugee*s. Because the threads changes their status rapidly and live observation does not mean observing frozen status, some types of reports on status change are not meaningful. As the number of threads increases, the non-blocking report transfer causes significant response delay both in the *debug client* and the *debug server*.

On the other hand, threads may be suspended by the debugger, or hit breakpoints, or hit steps, or may be blocked trying to synchronize with the breaked thread, or may be deadlocked. Otherwise, threads may be blocked by long system calls waiting for some input from outside. For these threads, "fine-grained" information on status should be shown. The information on status must include "code place" (the last execution place in script code when jumping into native code), holding lock objects, etc., so that the developer (user) can judge that that the threads are stopped as intended or not, e.g., correct in logic but wrong in scheduling, e.g., the false sharing problem. Therefore, while the *debug server* is required to update the fine-grained status of threads, the reports to the *debug client* are minimized.

[2]Ruby API has also a "synchronous termination"(Thread.kill). In both mechanisms, those in the *sleeping* state must be terminated by supplying input.

C.2 Asynchnonous Thread-aware Debugging Features

C.2.1 Concurrent Remote Debugging of Multiple Processes

Remote debugging, i.e., connecting and launching remote processes, is essential to debug communicating processes. We hereafter refer to the debugged process as "debuggee".

Being able to handle more than one *debuggees* is desirable, not only for user ergonomics by saving window space but, more important, for new (potential) capabilities such as remote stepping and session-through web application tracing.

C.2.2 Asynchronous Snapshot of Threads

Figure 10 shows the thread status change diagram we use, where the nodes indicate the status, and the arrows indicate the *events* on status change.

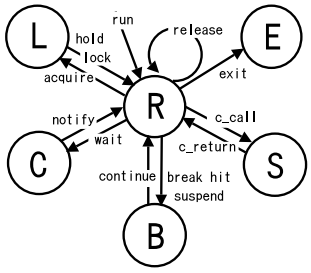


Figure 10: Thread status transition diagram.

Thread status in debugee view: Each leaf has an indicator to show the thread status followed by lock objects it is trying to hold (wants <object list>) and those it is holding (holds<object list>):

- *R (running):* The thread is either running or being ready to run. The thread may be holding one or more locks indicated in the optional *object list*.
- *E or E! (extinct):* The thread has terminated its execution normally (*E*) or by exception (*E!*). Holding some lock *object* may be a serious error.
- *B (break suspended):* The thread has hit a breakpoint or hit a step, and therefore, is waiting for commands.
- *L (locked):* The thread is blocked by a lock object in the "wants *object list*".
- *C (condition waiting):* The thread is blocked by a condition variable.
- *S (sleeping):* The thread is waiting for external input and is out of debugger control. Holding a lock objet may be problematic.

Receiving reports on status change in debug client: The *debug client* always reacts to the status reports coming from the *debug server* and update the *debugee view*. The *debug server* sends only key notification reports such as *break suspended*, *thread created*, and *terminated* immdiately when the events occur. Note that these events are much less frequent than others. The *debug server* updates the thread status, but postpone reports until a request comes from the *debug client*. The *debug client* requests for full status reports at contex switch time. Clicking the leaf symbol in the *debugee view* causes the command for "status (of the current thread)" to be issued, and clicking the node symbol, "statusList (of the current debugee process)"to be issued.

C.2.3 Low-intrusive Debug Operations

Break points per-thread and common to threads: We should support several kinds of break points: 1) per-thread, i.e., effective to only one thread; 2) common to all the threads, i.e., any thread that hits the break point, breaks, while other threads remain unaffected by the breakpoint hit; and 3) process as a whole, i.e., traditional break points.

Distrub-modes to control a set of theads: *Disturb-mode* facilitates users to cause some group of threads that are unknown or difficult to know to break.

- The initial idea of *disturb mode* was to specify new threads before their creation.
- One extension is *rmi-disturb mode*, which causes RMI (Remote Method Invocation) proxies to break.
- The other extension is *disturb-action* mode to enable and disable the automatic breakpoint setting for a specified session.

Controlling individual threads: A thread of interest should be suspended or resumed, or stepped sequentially, while other threads are running normally.

Plug-in and out of instrumentation code: Being able to plug in and out instrumentation code during debugging is necessary to monitor the real-time behavior of the *debuggee*. Such instrumentation code may be attached to any place of program code[3].

Interleaving debug sessions: User interactions to control individual *debugee* threads should be independent of each other. We call each interaction sequence "session". *Sessions* must be interleaved but should not be inter-mixed in user interface. The debugger, therefore, must accept one or more break or step event at the same time, and react for the session that the user wants.

C.2.4 Graphical User Interface(GUI)

We must facilitate the management and operation of involved processes and possibly many threads. GUI support is essential to enable users to grasp the whole status of debuggees, particularly that for the management of threads, and to switch interleaved *sessions*.

C.2.5 Multiple Language Support

Different languages are used in distibuted applications. Processes that execute code written in different languages may communicate with each other[4]. Dionea should be able to debug such processes together.

C.2.6 Portability for different platforms

Scripting languages are portable, therefore, Dionea should not be specific to any operating system or to any thread library[5].

D Domain Specific Feaures of Dionea

D.1 Automatic recognition and tracing of peer theads

D.1.1 Dealing with Peer Threads

As a pair of caller and callee threads can be regarded one logical thread, as shown in Fogure 11, Dionea enabled to hold them pairwise, to step from the former to the latter, and to backtrace their stacks.

Since client proxy threads may synchronize (or confilict) with each other on a server process, handling local peers is important.

[3]Extending *trace points* for a set of class methods is a part of aspect oriented programming (AOP).

[4]using YAML format as later introduced, or XML format in web services

[5]Python interpreter runs with POSIX, GNU P-th etc., thread librares. Ruby interpreter has its self-contained threading inside.

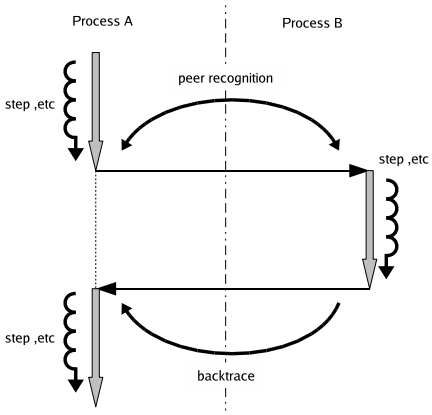


Figure 11: Tracing RMI.

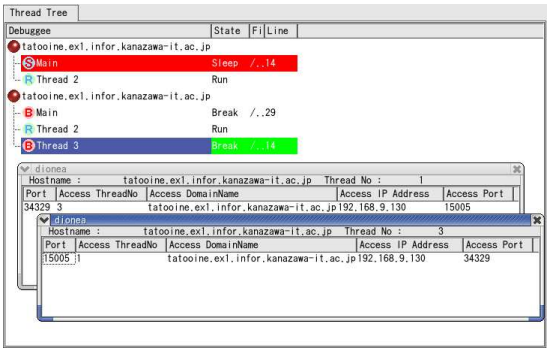


Figure 12: Displaying peer threads.

D.1.2 Automatic Peer Recogniction

Dionea deals with peer threads. It detects peer threads by using shared objects as key information. Such key information is reported by hooks that are plugged-in the debuggee at debug time. In the debugger, local peers are known by matching synchronization objects they share, while remote peers, such as caller and callee pairs using dRuby RMI (Remote Method Indication), are recognized by matching the pair addresses of sockets they use.

- As shown in Figure 12, in the debugee view, double clicking the thread (leaf) by right button, its remote peer thread is highlighted. Clicking the thread (leaf) by right button causes a popup view to show the socket information in details. Local peers are known also by this popup window.

- Selecting the callee side debugee, set the “disturb-rmi” by the popup menu in the toolbar. This causes all the newly created proxy threads on the callee side break at the callee methods.

D.1.3 Remote Stepping of dRuby RMI

Adding by hooks a step indication to messages enables stepping from a sender to a receiver and remote stepping from a caller to a callee of RMI in low-intrusive way. In the later case, the hook must judge whether both sides agree upon the hooks or not. We let the debugger attach “uri(s)” that identify the remote object in dRuby, to the step command.

Figure 13 on the left half shows an example of remote stepping. The stepping (in the upper half) causes a proxy thread to start and to break at the remote method entry, when the source code view is switched from the former to the latter (in the left side lower half). By a double click, both threads are highlighted

(right upper in the lower half). Other threads have no influence, though may compete with the latter for locking.

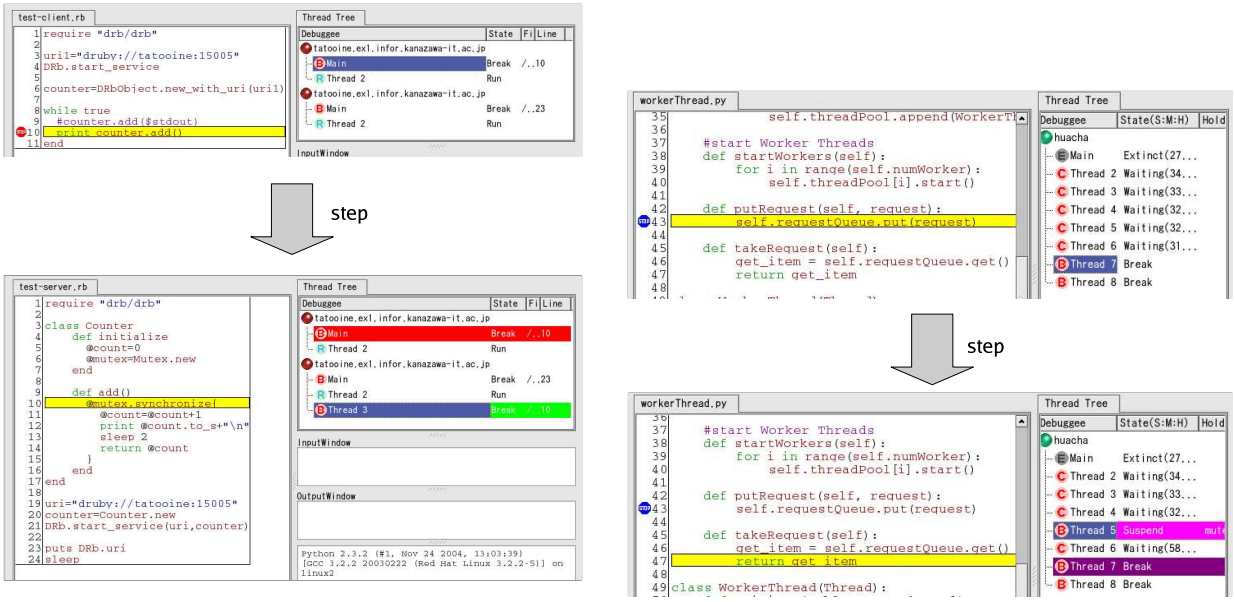


Figure 13: Remote stepping (left) and stepping from sender to receivers (right).

D.1.4 Tracing Asynchronous Message from Sender to Receivers

When peers are many senders and many receivers, one may want to trace the message flow via producer-consumer queue objects. Figure 13 on the right half shows an example. The solution to do this in a low-intrusive way is a subset of the RMI stepping, i.e., to add a "hidden step indication datum" preceding actual messages. This parameter is set true only when the sender is stepping, and cause the receiver to break.

D.2 Session-aware tracing of web applications (see Sec. 8)

Complex web applications are executed in an asynchronous event-driven way by individual web scripts that react to HTTP requests coming from a browser, but share the same data whose key is the "session ID" carried by the requests. Debugging their session-through behaviors and the interactions among sessions are difficult with thread-awareness alone, since in their execution platforms, i.e., web servers, one thread may be reused for more than one session and one session involves more than one thread possibly in different processes.

Therefore, we propose the following "session-aware" debugging features as deccribed in Section 8. This augmentation enables a seamless switching between automatic regression test and manual test/debugging.

E Effective Usages

E.1 Multi-thread Programs

Dionea acts for single and multi-threaded programs, since it has superset features of traditional debuggers. In the following, we present threading patterns for which the *low-intrusion* environment is effective.

Resource sharing pattern By causing perturbations in thread scheduling, mutual exclusion errors can easily be detected. For example, in the well-known "dining philosopher problem", even if we deliberately write an error such that the left-hand side and right-hand side forks are acquired separately,

dead lock state does not occur in a short time in normal scheduling. However, by simply stepping one philosopher thread letting others run, a dead lock appears immediately. When a philosopher acquires its right-hand side fork and stops by stepping, its right-hand side philosopher suspends its execution holding its right-hand side fork trying to acquire the other, etc., and eventually, every philosopher thread suspends forever holding its right-hand side fork. By giving scheduling perturbations to a great extent, we may very often detect hidden bugs caused by resource sharing errors.

Producer-consumer and Worker-thread patterns *Low-intrusion* environment enables to debug a *consumer (worker thread)*, while letting the *producer* get input from outside of the process. With *stop-the-world* environment, the input is also stopped while debugging the *consumer*. Thanks to the hook attached to mutual exclusion and conditional variables, the status change of the involved threads are made visible through the *debuggee view*. To know which thread is locked on which variable is easily known by switching the *thread context* and looking into the highlighted source code.

Thread-per-message pattern This pattern is similar to the producer-consumer pattern, except that the worker thread is created per message. In this case, the new thread must be suspended at the same time as its creation. No thread identifier for such a thread is available. Dionea provides with *disturb mode*, which causes a newly created thread to be traced and suspended. The *disturb mode* can be enabled and disabled by an icon in the tool bar. This one click feature is similar to the temporary break point but easier for the user to handle.

Read/write lock synchronization: One may step a reader to check all the writers are locked, or step a writer and to see all the peers are locked.

Barrier synchronization: One may hold a thread and see its peers get into locked state until it gets through the barrier.

E.2 Distributed Objects

The *debug client* manages connected processes and all the threads inside of them. With user's knowledge in the source code, this facilitates the debugging of distributed and multi-threaded programs. While processes run asynchronously, however, in some applications, thread pairs may be coupled by CORBA type query/answer RPC (Remote Procedure Call) as if "one virtual thread". It would be useful to enable the tracing of virtual threads across processes. "Pyro" for Python[6] and "dRuby" for Ruby are such platforms. The *debug server* could, like other hooks, add/remove some appropriate hooks to Pyro and dRuby at debug time. See Appendix D.1.

E.3 Server-side Web Applications (Rails and Turbogears)

Automation strategy: Reuse of test cases helps us to automate tests. With the Selenium-IDE add-on, the user operations on the browser can be recorded as a sequence of commands, which can be edited and replayed. This can be used to produce automatic regression tests. Note that a *debuggee* crash due to a bug in the application code does not force the developer to restart the debugging session, since session ID and session data are outside of the crashed process. Note also that dynamic page flow and concurrency can hardly be replayed with a simple sequence of commands. These are pros and cons to automate tests. Therefore, we propose to combine this add-on on the browser side with the augmented Dionea on the server side, and thereby to provide a semi-automatic environment that has a seamless capability to switch between automatic and manual tests. The following subsections give sample scenarios.

Automatic session-through test: Figure 14 shows a snapshot of the depot application test, for which we have added a visualized cart feature on the catalog page by using Ajax, and thereby saved one redirection. As shown in Figure 15, let us suppose that the user drags the book titles one by one into

[6]Not implemented.

the cart. These operations can be recorded and replayed in the Selenium add-on as a sequence of commands[7] as shown at the lower-right part, where "fast", "slow","replay", "step", "redo" and "record" buttons are provided as well. When *disturb-action-mode* is reset, the test proceeds automatically. The successful request and failed assertion commands are highlighted using different colors.

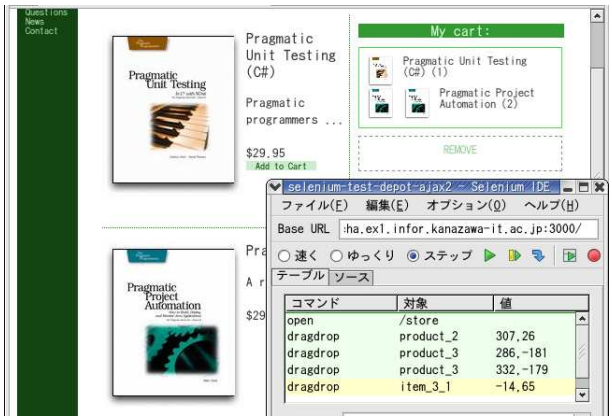


Figure 14: Debugging with Selenium IDE for FireFox.

Seamless switch to/from debugging: When an error is observed on the browser or an assertion command is violated, we should set *disturb-action-mode* and replay the commands, and thereby, switch to debugging. An Ajax call can be regarded as an HTTP request on the server side, therefore, the debugging steps are as follows:

1. Drag the thumb-nail of a product to the cart feature.
2. Drop it there, and action "StoreController#add_to_cart" breaks.
3. Complete the trace of the action, and the cart feature within the catalog page is re-rendered. We observe that the product has been added to the cart.
4. Reset *disturb-action*, and click the tag "show my cart", the script is executed without any break. The cart page appears on the browser, and we can confirm the contents of the cart.

Ajax call interaction: Since more than one Ajax call may come simultaneously, there may be a problem of locking "session data", e.g., the cart object. For example, if, after step 2, we drag another product to the cart, we get two breaks of web scripts at the same time. We can let one script go first, or interleave the two. In either case, we observe that one of the two has no effect on the cart on the server side, which does not agree with the cart figure. Selenium does not simulate the timing among user operations, and we cannot re-generate problems of this kind without manual intervention. One solution is to configure the server so that all the requests of the same session are executed by the same (backend) process, and not to use the thread-safe flag.

Automatic multiple session test: We have tried multiple session tests using three browsers with the Selenium-IDE add-on at the same time to observe the concurrency in web servers. Two browsers are used for the purchaser sessions and one for the administrator session. Figure 16 shows the *debuggee view* where all the threads created in WEBrick are displayed. Those that are not starting the execution of *web scripts* are hidden in the *session view*. Here, "Main" indicates the producer thread to accept incoming requests, while the rest are worker threads that execute web scripts. The upper section shows when scripts are assumed to be "thread unsafe" by default, while the lower section shows when the scripts are assumed to be "thread safe"[8]. In the upper section, among the three worker threads, we observe that the first thread runs and stops at a break, while the others are locked in the Rails library ("DispatchServlet" class). This does not mean that the whole WEBrick process is frozen. Only the

[7] In version 0.8.7, the "drag and drop" is not supported, but can be added using its extension features.
[8] ActionController::Base.allow_concurrency=true.

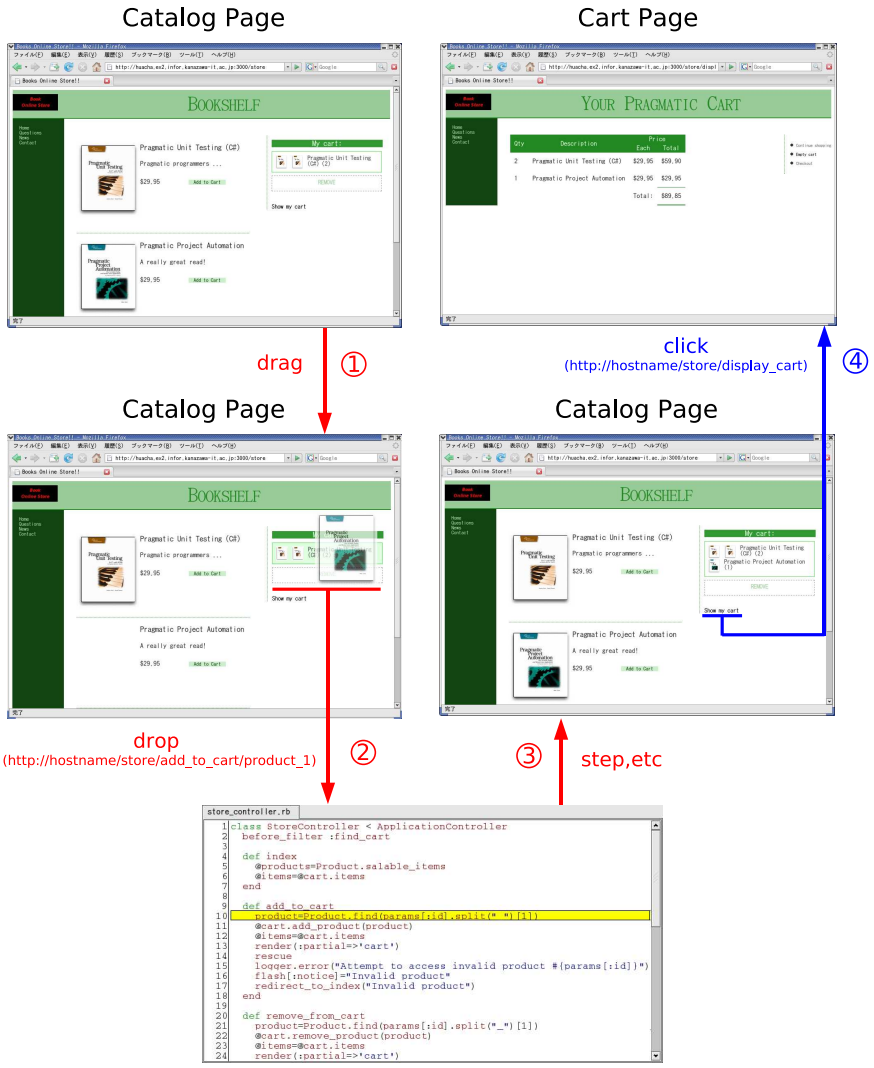


Figure 15: Debugging Ajax calls.

execution of the two worker threads out of three are locked in the Rails library, while the process itself is ready to accept new requests.

Figure 17 shows the *debuggee view*, where all the threads that are created in FastCGI processes are displayed. We observe that a new FastCGI process is created for each of the simultaneous requests within some upper limitation. The excessive requests if present are held in the process manager that runs in the frontend process of the FastCGI processes. We observe that Rails, by default, runs its application with single thread and scales to requests by processes. Moreover the debugger feature of handling more than one *debuggee* asynchronously has its full effect to trace multiple sessions simultaneously.

Testing for race conditions: Two administrator sessions could cause a race condition such as duplicated shipping. In reservation systems concerning tickets, trains, rooms, etc., the developer must provide a rollback page when a transaction or an optimistic locking fails, and thereby avoid error reservations such as over-booking[9]. While such a critical test is not often in regression test suite, there are cases where the page flow in a single session context may be different from that in simultaneous sessions, depending on the contents of database inventories. Moreover, it is unpredictable as to whether the user

[9]We do not know a clear and general solution of this kind of mission critical problems.

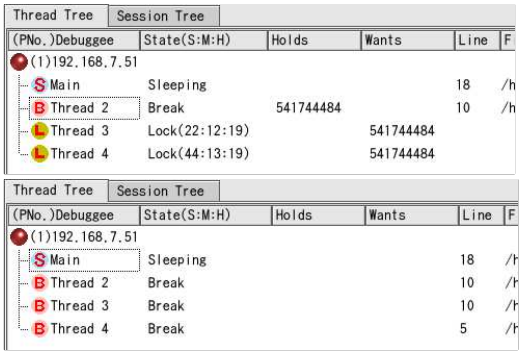


Figure 16: Concurrent execution in WEBrick.

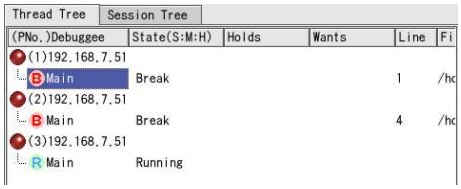


Figure 17: Concurrent execution in Fastcgi.

may attempt to move back-pages to try again. Even if we could extend the Selenium-IDE with control and loop commands, it would hardly be feasible to record all possible user reactions and therefore we need a manual test. The feature *disturb-action mode* assists the developer to re-generate such race conditions.

F Dionea Inside

As shown in Figure 2, we have split the debugger into *debug client* and *debug servers*, and coupled them asynchronously.

F.1 Debug Client

F.1.1 Basic architecture

GUI tool kit: The *debug client* must concurrently handle both GUI events caused by user commands and debug events coming from *debug servers*. As no "thread-safe" GUI tool kit is available for Python, we have to let one thread to handle both types of event. PyQt[10], which is a Python wrapper of Qt fullfills this requirement. PyQt provides with non-blocking API to incorporate TCP socket as one of its widgets and to handle its incoming messages as GUI events.

Context hierarchy and widgets:

Contexts are heirarchical. Switching a *process context*, the *thread context* is also switched[11], and vice versa. Each *context* has a set of GUI widgets. As shown in Figure 18, a process node and a thread node have a parent-child relationship. Each *thread context* node retains its own view sets, such as a source view and a command view. The difference between selected and not selected contexts is only that the former contents are shown in the window while the latter is not. The thread number attached to the thread (process) operation commands is obtained from the selected thread (process) context, by default.

[10] http://www.riverbankcomputing.co.uk/pyqt/
[11] Note that a *process context* has at least one *thread context* 'main'.

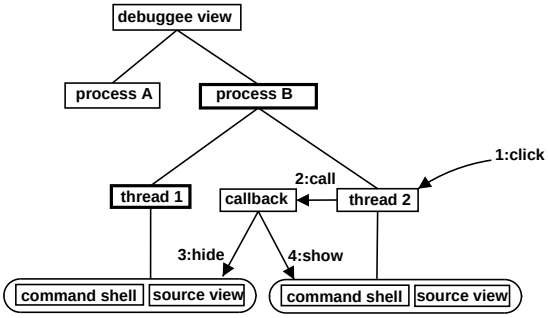


Figure 18: Widget architecture.

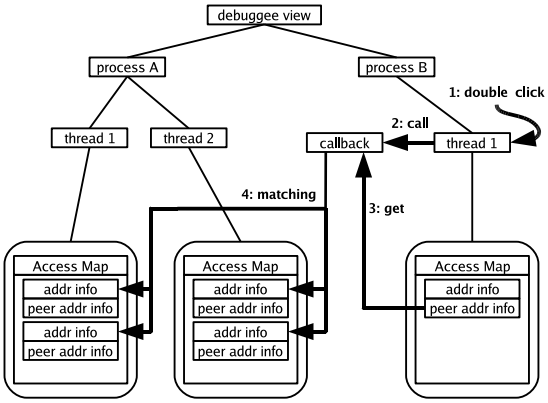


Figure 19: Remote peer recognition in the debug client.

Figure 18 shows also an example of context switch from hidden to current. In the debugee view, when the "thread 2"node is clicked, the "callback" function is invoked. This function hides the view set of "thread 1", and shows that of "thread 2"[12].

F.1.2 Peer Thread Recognition

As shown in Figure 19, two threads are peers when their *thread contexts* have the sockets that match with each other, or, as shown in Figure 20, share the same synchronization objects: For sockets, their properties (such as soket types and address pairs) must match; For memory objects, their object identifications must match.

F.1.3 Grouping Threads by Sessions

Using FastCGI, sessions are recognized as shown in Figure 22. Whichever process takes over a HTTP request, the hook in the router informs its thread and session, and the thread appears as a leaf of the session identifier in the *session view* as shown in Figure21. Observing the *thread view*, the user knows the thread and processes.

F.2 Debug Server Python

Figure 23 shows overall structure of the *debug server* for Python. The *debug server* code consists of three parts: thread trace module (ttc module)[13], a command listener, and callback functions. The *debug server* for

[12] At this time, "status" command is sent to the server side, and if necessary, update the status indication.
[13] to interact with CPython interpreter to get threads into traced mode, and to terminate threads.

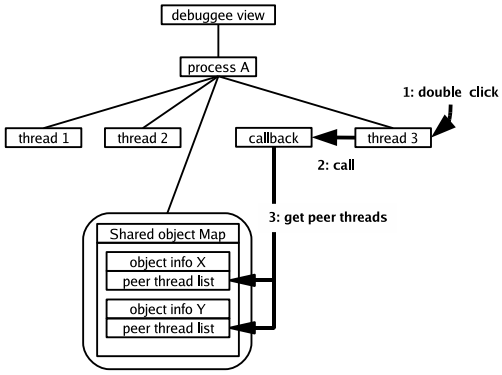


Figure 20: Local peer recognition in the debug client.

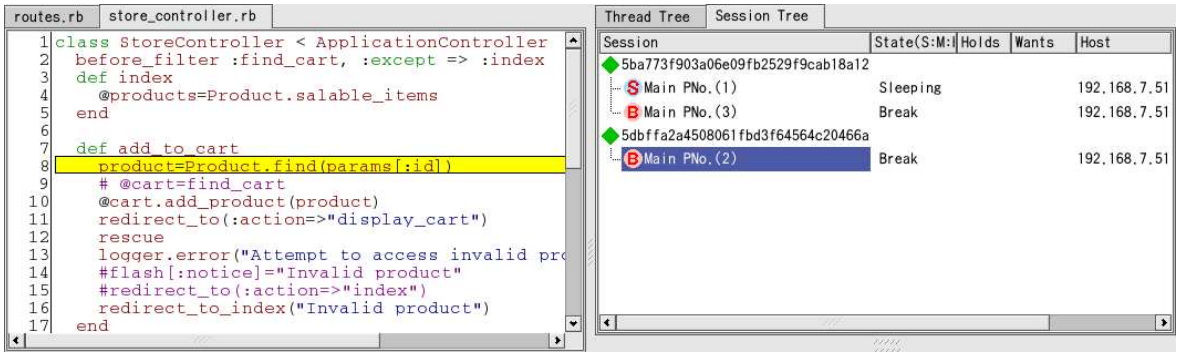


Figure 21: Session view for web application threads (using FastCGI).

the Ruby is quite similar, except for "ttc". The *command listener* is a dedicated thread for receiving debug commands and sending back debug events. For each *debuggee* thread, a "debug object" is created, which contains its intrinsic debug data. The debug data common to all the threads are shared among the *command listener* and *debuggee threads that execute callback functions*, and are accessed in a mutually exclusive way.

F.2.1 Trace-hook Reinforcements

Python debug API offers a hook named "settrace" that causes the specified Python function to be invoked (i.e.,piggybacked by the traced thread) at each time of one line execution, or a function call, or a function return, or an exception raise. Two problems were found and we solved them as follows:

1. First, the hook can be set only for "main thread", not for other threads, although the thread control blocks are listed inside of Python interpreter as in Figure 24. A method does exist to get the list pointer, so we have provided a new settrace interface that accept a *thread number*. We coded this part in C as an extension module we call "ttc (thread trace control module)", and made it a dynamic link module[14].

2. Second, no callback event is provided for thread creation and termination. Python API for thread creation is "Thread" class and its "run" method. We attached a "hook" here to generate an event at the entry and the exit of this method.

   In scripting languages, having symbol dictionaries at runtime, to plug-in and out such hooks can be

[14]Other parts are coded in Python itself.

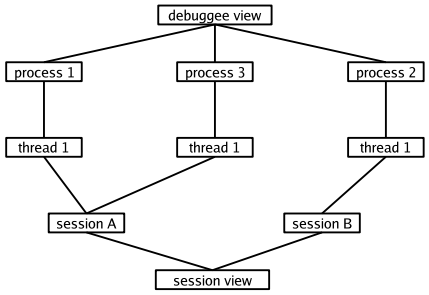


Figure 22: Grouping distributed threads by sessions (using Apache/FastCGI).

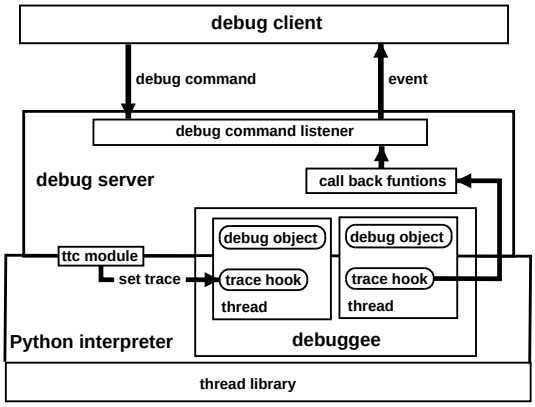


Figure 23: Debug server architecture.

done at runtime. The basic technique is to replace methods at debug time[15]. This is also done for generating "wait" and "lock" events for synchronization objects.

F.2.2 How the debug server start running

The *debuggee* is run by the *debug server* as follows:

1. The main thread of the *debug server* loads[16] and attaches hooks[17] at the thread libraries to generate debug events such as thread creation, termination, etc., that are not supported by a debug API callback by the interpreter.

2. Then, the main thread creates a dedicated (hidden) thread, called a "command listener", to receive debug commands and to send back responses. It then executes the specified *debuggee* script with its context as main. Thus, the *debuggee* thread(s) and the *command listener* run concurrently in the same process.

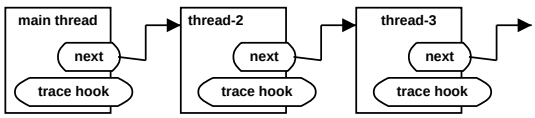


Figure 24: Thread state table.

[15]For Python, "hook module" provided by Twist (http://www.twistedmatrix.com) facilitates hooking some code at the entry and return of a method.
[16]Simply done by *require* in Ruby and *import* in Python.
[17]Simply done by the assignments to the methods, in dynamic languages such as Ruby and Python.

The user can let the *debug client* "connect", "disconnect", and "re-connect" to a running *debug server*, or to "launch" it.

F.2.3 Command listener

The *debug server* receives the incoming debug commands that the *debug client* issues and informs to it of the debug events such as break-hit and step-hit reports. Unlike traditional debuggers, commands and reports must be handled asynchronously. While executing a command for one thread, other threads must not be blocked. We need, therefore, a dedicated thread to receive commands and to send debug events, which we call "command listener".

As shown in Figure 25, the *command listner* handles the received commands as following:

Commands for setting a break point: A break point object is created, and, is registered, with the source code position as key, in a break point dictionary that resides either in the common *debug object*, or in the *debug object* that is intrinsic to the specified thread.

Command for suspending a thread: The thread intrinsic *debug object* is flagged so that the corresponding thread may be suspended by the next execution event of settrace.

Other commands: For each thread in the *debuggee* a "producer-consumer FIFO queue" is provided. The commands are "put" into this queue and the specified thread is notified to execute them.

F.2.4 Callback Functions

The callback function specified to the "settrace" hook is invoked piggybacked by the debuggee threads that are traced by the interpreter, and thereby, we can detect debug events. Figure 26 shows the flow of starting analysis. The callback function is passed the execution event (that is either "line", "call", "return", or "exception") and the current stack frame object (that contains the current context), as paremeters from the interpreter, and then, checks the needs for suspension.

F.2.5 Command Execution and Event Generation

Setting hooks: Hooks are instrumentation code to generate debug events on thread status change. We let the *debug server* plug-in code at the entry and the exit of built-in classes at connection time. We let it plug-out some of the hooks at disconnection.

Setting break points and trace points: As shown in Figure 25, they are set by the *command listener*, while other commands are executed indirectly via a producer-consume queue in the "debug object" by *debuggee* threads.

Trapping watchpoints and callback functions: As shown on the right-hand side of Figure 25, watchpoints are trapped by "settrace" callbacks as shown in Figure 26, which are piggybacked by the debuggee threads.

- *line (before the line execution):* is used to detect the hit of *line number* type watchpoints, *data watchpoints*, and steps. The attached instrumentation code if present is evaluated for "trace points". A data watchpoint is hit whenever the value of the specified *expression* is changed from its previous value.
- *call (after invocation):* is used to catch the "cutpoint" type watchpoints. The "before advice" type instrumentation code if present, is evaluated.
- *return (before return):* is used to evaluate "after advice" type instrumentation code if present.
- *exception (an exception is raised):* is used to report that the thread is terminated by some uncaught exception, when the exception is *sys_exit*, othewise is ignored[18]

[18] How to distinguish uncaught exceptions is under investigation. If possible, we should break the thread execution.

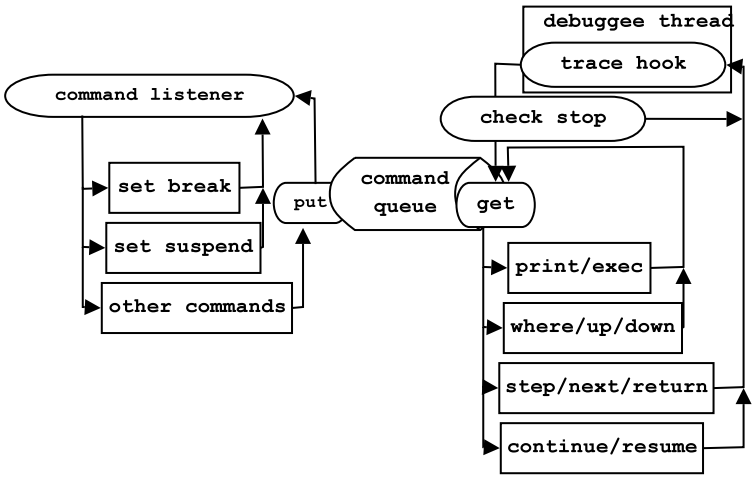


Figure 25: Command dispatching.

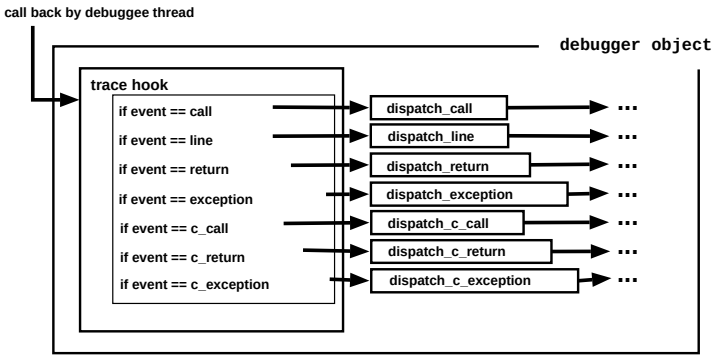


Figure 26: Callback by debuggee threads.

- *c_call (before calling native code):* is used to generate the status change event from *running to sleeping*. The lastly executed script code we call *place* is recorded, as the *settrace* control is being lost.
- *c_return (after returning from native code):* is used to recover the status change from *sleeping to running*.
- *c_exception (native code raised an exception):* is ignored for now.

When a breakpoint or a step is hit, the thread reacts to the command in the queue coming from the debug session in the *debug client*.

For example, when the event is *line*, the analysis is as follows: It checks if a suspension requirement is flagged, or, if the current frame is the frame of suspension for *step*, *next*, or *return*, or, if the line is a break point, then, the thread suspends itself and changes its status to *break suspended*.

Before suspension, an appropriate debug event is generated and buffered. Then, as shown in the right-hand side of Figure 25, the thread "gets" commands from its dedicated producer-consumer FIFO queue, and performs real operations, e.g., evaluates specified *expression*, until it gets *resume* or *step* or *next* or *continue* or *resume*.

F.3 Debug Server for Ruby

We implemented the *debug server* for Ruby, by rewriting that for Python. Two problems in Ruby debug API were found for our purpose:

1. Ruby provides with a debug API "settrace" similar to Python. But it causes all the threads to be traced by the interpreter. It does not allow for "thread by thread" tracing.

2. The stack frame object passed to the callback function lacks in dynamic frame link. Without this frame link, we cannot support stack frame *up* and *down*, unless the thread is traced from its starting time. This is a much more serious problem. The *debuggee* should always be traced even after disconnection, in order to be ready for reconnection.

The inside of Ruby interpreter however, cannot be accessed by some other extension module (hidden by using internal names) even if we use C. We understand that "hiding the inside of the interpreter" is the Ruby designer's policy, and expecting future extension, we used Ruby debug trace API as is. The problems are not fatal with functionality respect. The consequence, however, is that we cannot recover the normal execution speed of Ruby *debuggees* by disconnection.

F.3.1 Hooks to Generate Domain Specific Events

Hooks to remote stepping into peers in dRuby:

We use the fact that the first parameter of RMI is a method name string followed by arguments. We add a step indication parameter that is not a string preceding these parameters. However, this solution works only when both sides are running by the Dionea *debug server* and connected to the *debug client*, but this is not always the case. One safe and fast solution is to let the *debug client* inform the list of remote objects identified by "uri"(s) that are defined in other processes run by Dionea *debug servers*. We let the *debug client* inform such remote objects to the *debug server* at each time of stepping. Therefore, we attach such a list to the *step command*:

step [n] < thread-num><uri list>

From the user's point of view, this command is just *step* and optional number of steps n or single clicking *step icon*.

Figure 27 summarizes the hooks. The bold lines with arrows indicate the messages to and from the *debug client*. The hooks enclosed by '[' and ']' are those executed only at the first time of the remote method calls.

Matching uri to know callee side has the hook: (1) The "uri as definition" is reported to the *debug client* when the remote object is put in service, or when "connected" if it is already in service, and is held in the *process context* of the *debug client*. (2) The "uri as reference" is not reported to the *debug client*, but is used when the *step* command is sent: If the reference matches with one of the "uri list", (6) if the "uri" matches, the caller side hook attach a flag parameter to the message, otherwise, as the callee side has no hook to accept it, nothing is done, and the *step* behaves as a *next*. (7 and 8) Detecting this flag, the callee thread suspends and hit a step at the mothod entry.

Makeing automatic context switch happen: (3, 4 and 5) when a TCP connection is established, the caller thread and the callee thread send the socket information they use to the *debug client*. (6, 9 and 10) To make the context switch in the *debug client* happen automatically, the hooks send debug events telling a remote call, and (11) remote return happened, respectively.

These hooks are plugged-in at connection and plugged-out at disconnection.

Hooks for Local Peer Recognition and Tracing:

The solution to do this in a very low-intrusive way is a subset of the RMI stepping, i.e., to add a "hidden step indication datum" preceding actual messages. This parameter is set true only when the sender is stepping, and cause the receiver to break.

Hooks to get Session ID:

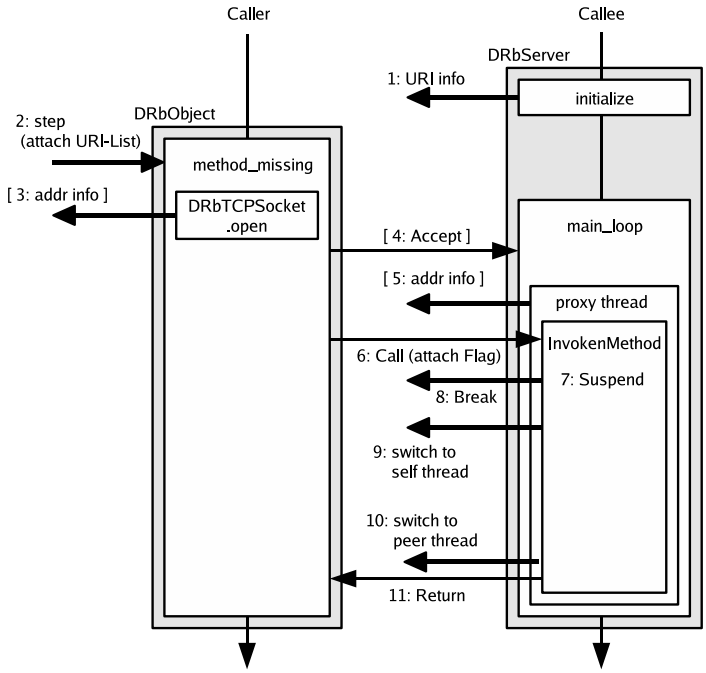


Figure 27: Hooks in dRuby RMI.

We let the *debug server* to set a hook at the Ruby standard library[19] to inform the *debug client* of the session identifier (i.e., cookie), as shown in Figure 28.

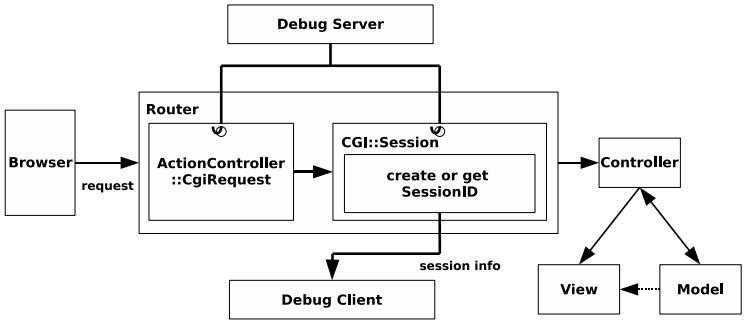


Figure 28: Hooks to get session ID.

Hooks to set breakpoint at controller action: In Rails, request URIs are mapped to methods of controllers, called *actions*, for which rules are specified by configuration. We let the *debug server* to set "hidden breakpoints" to such actions on-the-fly. As shown in Figure 29, we let the *debug server* set a hook at the Rails library, where the mapping from a URI to an action is done, to generate such temporary breakpoint, but only when *disturb-action mode* is enabled. Such breakpoints are temporary, i.e.,effective only once.

For non-Rails web applications, this hook is not hit. Each web script breaks at it staring point by means of *disturb-mode*.

[19] the constructor of CGI::Session class

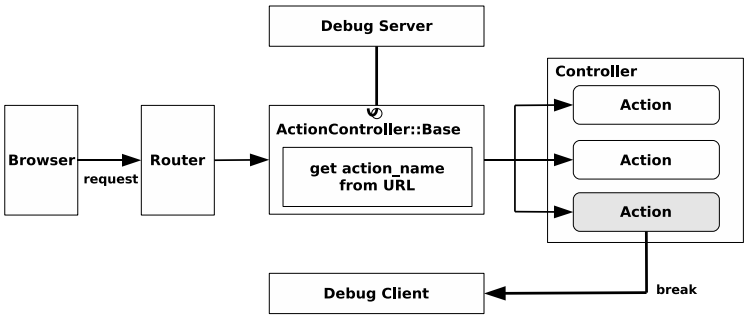


Figure 29: Hooks to set breakpoints automatically.

F.3.2 Controlling Disturb-action Mode

The *disturb-action mode* is enabled and disabled in the following way:

- **One-touch on and off:** As shown in Figure 6, as with the other disturb modes, enabling and disabling can be specified by a one-touch operation. This causes the following command to be issued from the *debug client* to all the *debug servers*:
  `>set-disturb-action-mode`
  `true/false` "`session ID`"
  where "session ID" indicates the currently selected session.

- **Propagation to a newly created debuggee:** The *disturb-action mode* must be propagated from one debuggee to another. To do this, we have extended the *connect* command:
  `>connect` "`session-info list`"
  where "session-info list" is one or more pairs of session ID and flag. The session ID is one that has been recognized by the *debug client*, and the flag indicates whether the *disturb-action mode* is enabled.

- **Initial mode setting:** Figure 30 shows the flow when the disturb modes are initialized in the *debuggee* by the *debug server*. For a session known to the *debug client*, the initial mode is either *disturb mode* enabled or *disturb-action mode* enabled. For an unknown session, *disturb mode* is always enabled. For the sessions of Rails applications, the initial mode may be either *disturb mode* enabled or *disturb-action mode* enabled.

Note: If we could exclude non-Rails applications, the inital mode would always be *disturb-action mode* enabled. This strategy is much simpler.

F.4 Client Server Protocol

The infrastructure of *debug client* and *debug server* communication is asynchronous. We used non-blocking TCP socket, not the query/response type RPC synchronization. The *command listener* uses "select" system call[20] for non-block sockets to wait for incoming commands from the *debug client*, and at the same time, to send debug events/data back to it.

To enable the *debug client* to communicate with *debug servers* for different languages, we need an interchangeable format among different languages to send and receive debug commands, debug events and data. As no existing XML based marshalling implementation is common to different languages, we use YAML (Yaml Ain't Markup Language[21]) that fulfills this requirement. YAML is a human readable document format for such data structures as "lists (arrays)" and "dictionaries (hashes)" of Python, Ruby, Perl and Java module, and marshaling/unmarshalling libraries are available for these languages. Figure 31 shows an example of YAML format corresponding with a Python dictionary, which is a command message for Dionea.

[20] "asyncore" module is used.
[21] http://yaml.org/

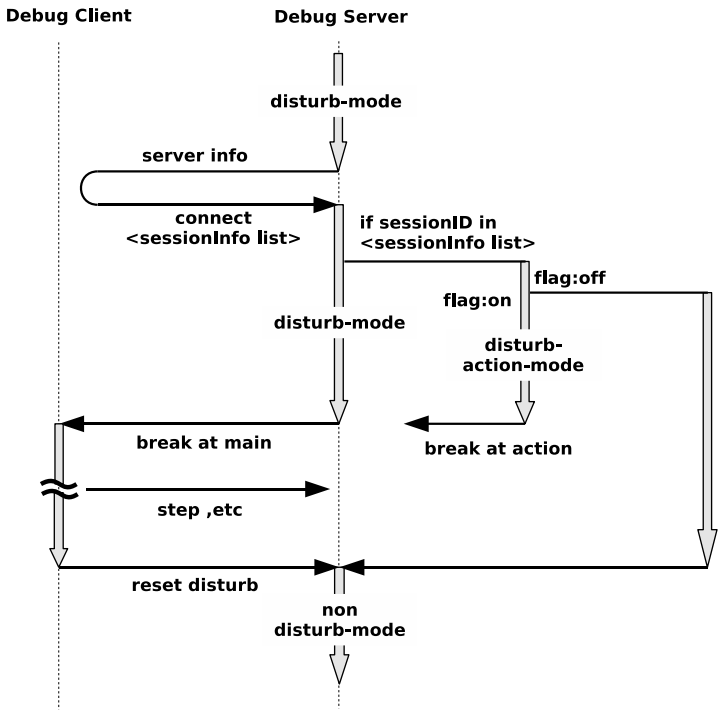
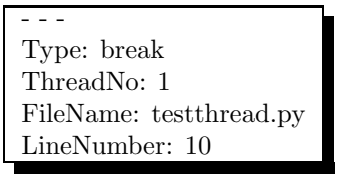


Figure 30: How disturb-modes are initialized.



YAML document

↓ ↑

{'Type': 'break',
'ThreadNo': 1,
'FileName': 'testthread.py'
'LineNumber': 10}

Python dictionary

Figure 31: Message transformation example.