Regular Paper

Low-intrusion Debugger for Python and Ruby Distributed Multi-thread Programs

Norio Sato,† Kazuhiro Nagai,† Yasushi Itoh,†
Masamitsu Ogura† and Keisuke Kosuga†

Major scripting languages such as Perl 5, Python and Ruby, provide with multi-thread features that could improve response time as for network programming, etc. We propose a new type of thread-aware debugger that provides us with "low-intrusion" debug environment. In the "high-intrusion" (or "stop-the-world") environment that existing thread-aware debuggers provide with, we must suspend the whole execution of a debugged process at each time of debug operations and responses. In contrast, the "low-intrusion" environment enables us to test the behavior of individual threads while other threads remain unaffected, which makes it possible to test a process under scheduling perturbations and workload in operation. We present the features and implementation of a debugger we have developed for Python and extended for Ruby. The debugger consists of client and server parts to handle communicating processes via network. Both parts are coupled with asynchronous messages encoded in a common format. This allows for developing the client part in common, while the server parts are implemented for individual languages. The client part can catch more than one process at the same time, and provides users with full GUI support to facilitate the handling of multiple threads inside of the processes. We have implemented the server parts with extended modules: trace-hook reinforcements, a dedicated thread that is listening to debug commands coming from the client part, and functions call-backed by individual debug-gee threads. We show the applications are wide: multi-thread programs, distributed objects, and Web programming. We present typical threading patterns for which low-intrusion is effective. We compare low- and high-intrusion environments, and propose an integration of both environments together with real-time tracing capability.

1. Introduction

Real-time programs such as concurrent servers, distributed objects, web services and web-server programs often run as communicating processes distributed among different computers. Such processes exchange asynchronous messages with each other, while each process executes multiple threads for interleaving requests and responses. Recently, in this area where performance bottleneck is network latency rather than execution speed of machine instructions, scripting languages such as Python [6),7)] and Ruby [13)], thanks to their high productivity, are coming to wider use [8),14)].

Whatever languages are used, this increases scale and complexity as a whole, and results in program code prone to hidden bugs that are hard to detect. In this "asynchronous world", debugging tools are few while the needs are many [1)]. One effective approach is to use debuggers for wider stages of development, from unit test, integration test, stress test to operational test.

Debugging distributed and multi-threaded programs needs much more efforts than single threaded programs. The difficulty to detect synchronization errors or resource conflicts is due to the non-determinacy of concurrency, and more practically, due to the difficulty to test with scheduling perturbations.

As an effective feature, we propose "low-intrusion" debug environment that enables us to check the behavior of individual threads, by break, step, trace, resume and suspend, etc., while the execution of other threads remain unaffected. This conceptually opposes to the traditional environment [2)] we call "stop-the-world" or "high-intrusion" environment, where the debugged process is suspended each time of command input and event response.

The "low-intrusion" environment enables us to cause scheduling perturbations at debug time, which offers us many chances to detect synchronization error code. It enables us to debug a process without stopping its interactions with outside world so that the debugged process may have the working load by receiving inputs and sending outputs as in operation. It enables

† Department of Information and Computer Science, Kanazawa Institute of Technology

us to input commands while a debugged process is running. None of these is possible in "high-intrusion" environment.

The "low-intrusion" environment is one of the fundamental concepts for the multi-thread and multi-process oriented interactive debugger we have developed, which we hereafter refer to as "Dionea"☆1.

The representative interactive native code debuggers such as "GDB"[3]☆2 and "DBX"[5]☆3 support more or less multi-threading enhancements with the traditional "stop-the-world" environment. The Java language debugger "jdb"is very close to DBX☆4. The existing debuggers for scripting languages are either unable or insufficient to test multi-threaded programs. For example, "Pdb" for Python works only for single ("main") thread. For Ruby, "Rdb" and a remote "Ruby debugger"☆5 do exist and have support for multi-threading, but are *highly intrusive*.

The first support language of Dionea is Python that is simple but has advanced features suitable for large scale distributed systems, as well as many other applications. Python has multi-threading API in an object oriented way[9]. Python is convenient for our research work, because its source code for scripting engine (interpreter) with many supportive modules☆6 is completely open and free. Next, we have extended Dionea to support Ruby. Ruby offers more advanced constructs than Python with its full-featured object orientation, iterators (i.e., blocks invoked by 'yield'), and threading constructs, with many supportive library modules[13].

This paper presents the features and implementation of Dionea, and discusses about low-intrusiveness and applications. In the following, Section 2 describes the design requirements. Section 3 describes characteristic features of Dionea. Section 4 describes the implementation. Section 5 presents how to apply Dionea to some typical network and multi-thread programs. Section 6 discusses about the low-intrusiveness in time and range, and evaluates the features that Dionea supports.

2. Requirements

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☆7.

2.2 Low-intrusive Debug Environment

Break points per-thread and common to threads: We should support several types 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.

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☆8.

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 events at the same time, and react for the session that the user wants.

☆1 Dionea means "Venus fly catcher", a plant whose academic name is "Dionaea Muscipula."

☆2 GNU debugger for UNIX recognizes Light-Weight Processes, which is not always "user threads" in Solaris.

☆3 For Solaris, Sun Micro proprietary.

☆4 Both DBX and jdb are thread-aware, and can easily suspend a thread, but cannot handle asynchronous debug events.

☆5 This works on top of "dRuby (Distributed Ruby)"[15]. It catches only one process at the same time.

☆6 http://www.python.org/

☆7 e.g., automatic tracing of spawned processes, synchronizing related events coming from different *debuggees*.

☆8 Extending *trace points* for a set of class methods is a part of aspect oriented programming (AOP).

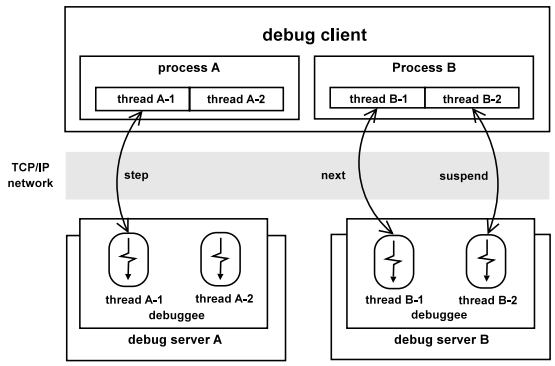

Fig. 1 Dionea architecture.

2.3 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*.

2.4 Multiple Language Support

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

2.5 Portability for Different Platforms

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

3. Proposed Debugger

3.1 Basic Archtecture

As shown in **Fig. 1**, in Dionea, the user interface and the debug operation parts are separated. We call the former "debug client", and the latter "debug server". This separation enables remote debugging one or more communicating processes with one debugger.

3.2 Running and Connecting Debugee

A *debuggee* is run by the *debug server* as below independently of the *debug client*, or through a dialog window of the *debug client*.

```
$dds.py 10000 /path/to/test.py&
```

The *debug client* is run by the command as below, and, through its dialog window, can connect a *debuggee* that is already run as above, or can (remote) run a *debuggee*. When run, the main thread in the *the debuggee* suspends waiting for debug commands.

```
$ dionea &
```

The *debug client* can disconnect and let the *debuggee* continue to run normally, leaving no debug side-effect. It can reconnect the *debugee*, and force it to terminate its execution.

3.3 Graphic User Interface (GUI)

Dionea interfaces with users by means of GUI, as shown in **Fig. 2**. The GUI consists of five 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 status of the thread. The status is either "running (R)", or "debug suspended (break) (B)" or "locked (L)"[☆☆☆]. The status indication is modified dynamically, when a change is detected in the *debugee*. We need the information of to which thread (or process) for the the *debug server* to operate commands and to which thread (or process) for the *debug client* to show its *session* contents in the window views. We must also retain a set of debug events for each *session*. We call the set of information belonging to a process "process context" and that belonging to a thread "thread context". A *process context* has one or more *thread contexts*. A *thread* (*process*) *context* can be switched from one to another by clicking a leaf (node), which causes the contents of other views that are described in the following to change.

Source code view (left hand side): By default, the suspended place of the selected thread is highlighted on the source code file. The source code file can explicitly be selected using the "source code tree" in an icon of the tool bar (upper left).

Input and output views (right-hand side middle): These views offer the standard input and output of the *debugee*, and is a part of *process context*.

Command shell (right-hand side lower): This view offers Command User Interface (CUI) to input commands and receives the output of the *debugee*, and is a part of *thread context*.

Tool bar (upper): Less frequently used

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

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

[☆☆☆] For Ruby, "sleeping (S)" is distinguishable from "running".

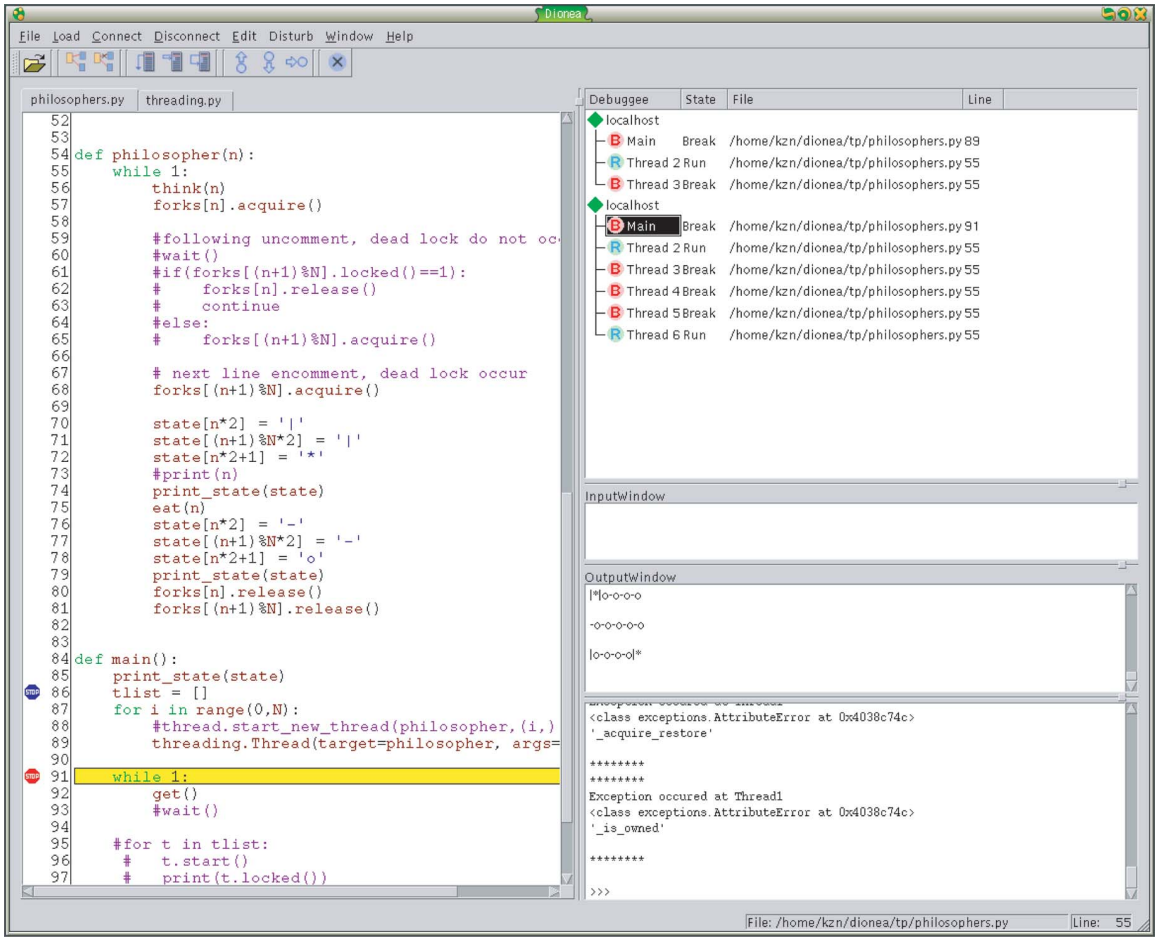

Fig. 2 User interface.

views, dialogues and frequently used commands are made icons. The former are source code file tree which is a part of *process context*, and connection disconnection dialogue. The latter are icons for *step*, and *up/down* operations. The "disturb" mode icon is for catching a thread that is being newly created.

3.4 Commands and Icons

Getting source code files: The source code for scripting languages always resides in the computer where the *debuggee* runs. The source code tree is built at connection time using the load-path directories of the *debuggee*. The contents of source files are obtained on-demand and cached. Source code requesting commands are issued implicitly when a node of the source code tree is clicked, and when needed to show the *thread context*. Since scripting languages allow for source code modification at debug time, we will enable the renewal of *source view* by explicit user requests. This issue, however, may involve break point renewal, and is left for further study.

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.

Thread operations:

"settrace <thread number>" requests the interpreter to trace the specified thread. A thread runs normally before this command is operated. This command is issued when the thread is clicked for the first time in the *debuggee view*. The thread number is taken from the selected *thread context*.

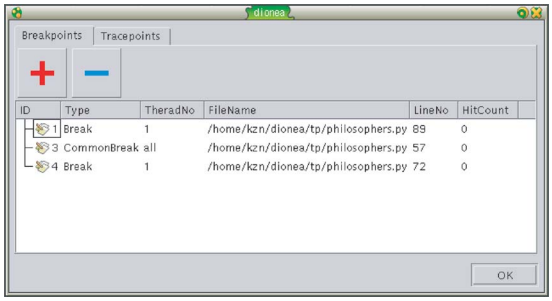

Fig. 3 breakpoints dialog.

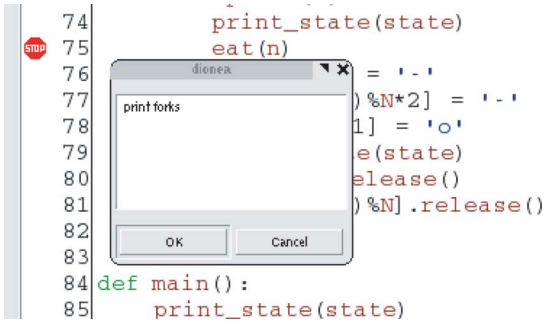

Fig. 4 tracepoints dialog.

"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*.

Break and trace points and their operations: The commands for setting break and trace points are not simple. Therefore, they can be set, changed their nature, and deleted by transitively clicking on the lines in *source view* (see Fig. 2), or by using the dialogs as shown in **Figs. 3** and **4**.

- 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.

Break points can be "enabled" or "disabled" or "deleted".

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*.

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 (i.e., 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. At present, no GUI commands are available for these commands, but displaying the values in a variable table is under study.

High-intrusive operations: "suspendProcess" and "resumeProcess" require all the threads in a process or grouped threads to be suspended or resumed. At present no GUI is available.

4. Implememtation

As shown in Fig. 1, we have split the debugger into *debug client* and *debug servers*, and coupled them asynchronously.

4.1 Debug Client

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

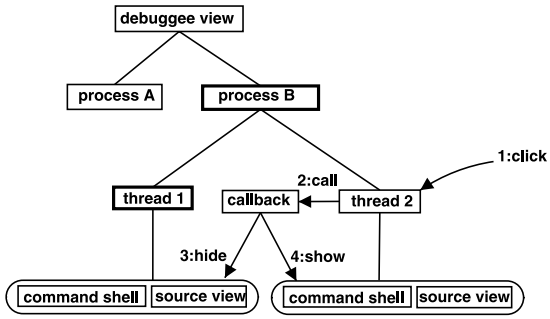

Fig. 5 Widget architecture.

types of event. PyQt[☆1 12)], which is a Python wrapper of Qt[11)] 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[☆2], and vice versa. Each *context* has a set of GUI widgets. As shown in **Fig. 5**, 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.

Figure 5 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"[☆3].

4.2 Debug Server for Python

Figure 6 shows overall structure of the *debug server* for Python. The *debug server* code consists of three parts: trace-hook reinforcements, a command listener, and callback functions. 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.

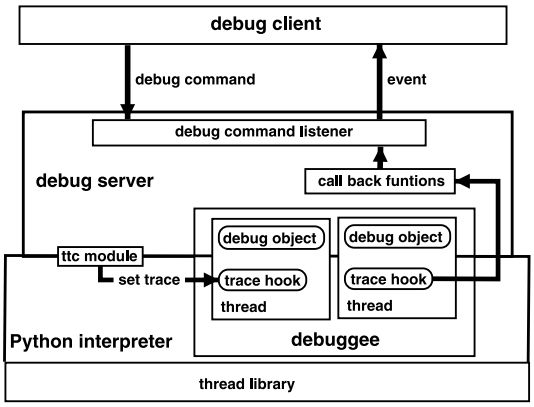

Fig. 6 Debug server architecture.

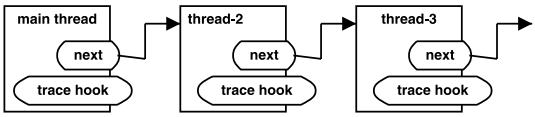

Fig. 7 Thread state table.

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 **Fig. 7**. 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[☆4].

(2) Second, no callback event is provided for thread creation and termination. Python API for thread creation is "Thread" class and its "run"

☆1 http://www.riverbankcomputing.co.uk/pyqt/

☆2 Note that a *process context* has at least one *thread context* 'main'.

☆3 At this time, "status" command is sent to the server side, and if necessary, update the status indication.

☆4 Other parts are coded in Python itself.

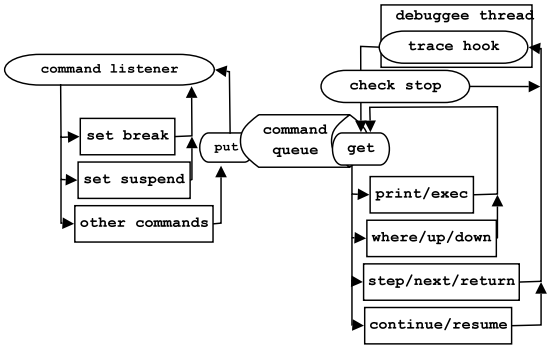

Fig. 8 Command dispatching.

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

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 **Fig. 8**, 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[9]" is provided. The commands are "put" into this queue and the specified thread is notified to execute them.

☆ For Python, "hook module" provided by Twist (http://www.twistedmatrix.com) facilitates hooking some code at the entry and return of a method.

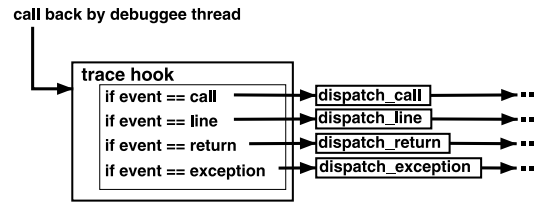

Fig. 9 Call back by debuggee threads.

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 9** 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.

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 Fig. 8, 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*.

4.3 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.

4.4 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☆ 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☆☆) 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 10** shows an example of YAML format corresponding with a Python dictionary, which is a command message for Dionea.

5. Applications

We are implementing some testing programs to evaluate the effectiveness of Dionea, categorized as following.

☆ "asyncore" module [9] is used.
☆☆ http://yaml.org/

```
- - -
Type: break
ThreadNo: 1
FileName: testthread.py
LineNumber: 10
```

YAML document

↓ ↑

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

Python dictionary

Fig. 10 Message transformation example.

5.1 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 [16] 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.

5.2 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" [10] for Python and "dRuby" [15] for Ruby are such platforms. The *debug server* could, like other hooks, add/remove some appropriate hooks to Pyro and dRuby at debug time. This will be a big work and is left for future study.

5.3 Server-side Web Programming

Being single or multi-threaded, CGI is one of the major application areas of Python and Ruby. Interactive debuggers are hardly usable to debug CGI (Common Gateway Interface) programs, due to the complexity of giving its running environment☆. We can let the "mod_python" for Apache☆☆, where Python interpreter is embedded, run the Dionea *debug server* for Python☆☆☆. To do this, we only have to write a configuration file of Apache in our private directories. Using well-known browsers and connecting Dionea "debug client", it proved that we can interactively debug CGI code, even while Apache with mod_python is in real operation. The same will be easy for the "debug server for Ruby" and "mod_ruby". Our next future support will be Perl that has much larger user population particularly in this area.

☆ Many environment variables must be set to run CGI for debugging.

☆☆ Apache version 2, which support multi-threading.

☆☆☆ Precisely, we added a few lines of "contents handler" code in the *debug server*, where the standard input and output is directed to Apache, not to Dionea.

6. Discussions and Eveluation

6.1 Intrusiveness in Terms of Suspension Time

Dionea is designed so that the execution of debuggee threads may never be suspended unless the user explicitly uses high-intrusion commands. However, as Dionea uses line-by-line callback, the overhead to trace a thread is not small. Using a small fraction of loop coded in Python, we observe slower speed by two orders of magnitude compared with the normal interpretation of "virtual instructions". Considering the execution of native modules and network latency, and the fact that multi-thread programs by nature are bound to input and output, this will be much less, say, by one order magnitude less, depending upon CPU performance. We observe no significant influence to thread scheduling under Linux on today's personal and note computers.

In contrast, we regard such process suspension as to allow human to input commands or to synchronize processes via network for command interpretation as "intrusion", which causes three orders of magnitude latency (from several tens of mill-seconds to seconds). In the sense that it does not cause this kind of suspension, Dionea is low-intrusive.

6.2 Intrusiveness in Terms of the Number of Threads

Dionea causes no overhead to non-traced threads. This is completely true for Python threads. Ruby threads cause all the threads the callback overhead, but still Dionea does not cause suspension. By step, break, and suspension, however, Dionea causes individual threads *high-intrusion*, which may not always be desirable. When the thread is time critical or when threads are tightly coupled, one thread suspension may cause the suspension of many other threads.

We therefore provide with *trace points*, which cause some instrumentation code (*expression* or *string*) to be evaluated with some overhead but without causing significant disturbance to its execution. The resulting values are buffered being sent back to the *debug client* unless spilt from buffering capacity.

6.3 High-intrusion vs. Low-intrusion

Considering the debugging or monitoring while the debuggee is running, the *low intrusion* is effective. However, in some critical situation where freezing global variables is needed, *high-intrusion* environment will be required. Therefore, Dionea combines both environments by providing with process level breaks and suspension.

7. Conclusion

We have proposed an interactive debugger that uses asynchronous one-to-many coupling of *debug client* and *debug server* architecture. This architecture enables *low-intrusiveness*, multiple processes debugging, and multiple language support. We have shown that its application area is wide: single and multi-threading, concurrent servers, distributed objects, and web programming. Our development so far is for Python and Ruby that are major advanced scripting languages suitable for these applications, and Dionea itself is one of them. Dionea by now runs on Linux, Free BSD and Mac-OS X.

While Dionea is a debugger for scripting languages, this architecture will in theory be implementable for native code debuggers. The native code debuggers today are high-intrusive, because they operate debuggee processes from outside. For example, UNIX debug API is "ptrace" or "/proc"☆. The debugger process (such as GDB) traps a modified machine instruction via signal☆☆ or receives callbacks☆☆☆ from outside of the debuggee process. However, if we could embed their debug operation code in the debuggee (as a dynamic link library), the low-intrusion features would be hopeful. Unfortunately, in reality, the feasibility is highly dependent on operating system and thread library supports, which are different from system to system or sometimes even insufficient.

We have implemented Dionea by making the most of the high productivity and the flexibility of Python and Ruby. The productivity means their high level nature and their many supportive modules. The flexibility means the runtime nature that allows for plug-in and out hooks at debug time, and the direct evaluation/execution of symbolic code. The trace points can be managed in symbolic forms, e.g., regular expression form. Though we have not yet done, we can insert any instrumentation code at the entry and exit of a group of methods. This will allow for AOP (Aspect Oriented Programming)-like features.

The performance decrease in Ruby programs debugged by Dionea disappears with the provision of per-thread tracing and dynamic link of frames, and therefore, is not an intrinsic problem.

Acknowledgments This paper is based upon our presentations in SIG Programming of IPSJ, in March 2003 and March 2004. We appreciate active discussions and suggestions there. We thank Mr. Trond Borsting and Mr. Harald Botnevik for discussions, comments and suggestions to improve the draft of this paper before submission.

☆ Memory mapped files, whose structures are different from UNIX to UNIX.
☆☆ SIGTRAP
☆☆☆ "libthread_db.so" for Solaris, which is not well documented.

References

1) Donat, M. and Chalk, S.: *Debugging in Asynchoronous World*, Vol.1, No.6, pp.23-30, *ACM Queue* (Sept. 2003).
2) Rosenberg, J.B.: *How Debuggers Work*, Wiley Computer Publishing (1997).
3) Stallman, R.M., Pesch, R.H. and Shebs, S.: *Debugging With Gdb: The GNU Source-Level Debugger*, GNU Press (2002).
4) Butenhof, D.P.: *Programming with POSIX Threads*, Addison-Wesley (1998).
5) Lewis, B. and Berg, D.J.: *Multithreaded Programming With Pthreads*, Prentice Hall (1997).
6) Lutz, M.: *Programming Python* (2nd ed.), O'Reilly & Associates (2001).
7) Acher, D. and Lutz, M.: *Learning Python*, Addison Wesley (1993).
8) Martelli, A. and Ascher, D. (Eds.): *Python Cookbook*, O'Reilly & Associates (July 2002).
9) David, M., Beazley, F. and Samarin, A.: *Python Essential Reference*, 2nd ed., New Riders (2001).
10) de Jong, I.: *Pyro Manual.version 2.1* (Oct. 3, 2001).
11) Dalheimer, M.K.: *Programming Qt* (2nd ed.), Oreilly & Associates (2002).
12) Rempt, B.: *Gui Programming With Python: Using the Qt Toolkit*, Commandprompt (2002).
13) Matsumoto, Y., Thomas, D. and Hunt, A.: *Programming Ruby: The Pragmatic Programmer's Guide*, Addison Wesley (2000).
14) Maeda, S., Matsumoto, Y., Yamada, A. and Nagai, H.: *Ruby Application Programming* (written in Japanese), Ohmsha (2002)
15) Seki, M.: *Distributed Object Programming by*

dRuby (written in Japanese), ASCII (2001), http://www2.biglobe.ne.jp/~seki/index.html.
16) Lea, D.: *Concurrent Programming in Java,* (2nd ed.): *Design Principles and Patterns*, Addison-Wesley (2000).

(Received March 29, 2004)
(Accepted September 3, 2004)

Norio Sato was graduated from Department of Mathematical Engineering and Information Physics, the Univeristy of Tokyo in 1972, worked for NTT Telecommunication Laboraories 1972–1998, and for Lucent Technologies Japan 1998–1999. He received Dr. Eng. from Ritsumeikan Univeristy for his thesis "A Study on Compiler and Testing Environment for Large Real-Time Telecommunications Software" 1999. Since 1999, he is a professor of Information and Computer Science, Graduated School, KIT (Kanazawa Institute of Technology). One of his hobbies is learning European languages, and recently Chinese. An outcome is "Learning Chinese Sounds through Three Dimensional Computer Graphics". He is a member of IEICE, IPSJ, and ACM.

Kazuhiro Nagai has M.Eng. from Department of Information and Computer Science, Graduate School, KIT, 2004 and now is working for NS Computer Services, Ltd. He got Excellent Student Prize from Hokuriku Affiliate of IPSJ, 2003. His interests are Linux, programming languages, compilers, tools and network software technologies.

Yasushi Itoh has M.Eng. from Department of Information and Computer Science, Graduate School of KIT, 2004 and now is working for Fuji Zerox, Ltd. He got Excellent Student Prize from Hokuriku Affiliate of IPSJ, 2004. His interests are Linux, FreeBSD, OS-X, and programming languages particulary SCHEME.

Masamitsu Ogura entered Department of Information and Computer Science, Graduate School of KIT in 2004. His interests are Linux and language processing technologies.

Keisuke Kosuga entered Department of Information and Computer Science, Graduate School of KIT in 2004. His interests are Linux and language processing technologies.