SystemTap and Java

1 Introduction **2**

2 Java probes **2**

3 VM – stpd communication interface (CI) **3**

3.1 Data streaming interface 3
3.2 Communication interface transport 3
3.3 CI details 4
3.3.1 VM agent to stpd flow 4
3.3.2 stpd to VM agent flow 4
3.3.3 strings and optional data placement 4

4 Compiler **4**

4.1 Java probe syntax 4
4.1.1 Java related structures 4
4.1.2 Methods 5
4.1.3 Classes 5
4.1.4 Exceptions 5
4.1.5 VM events 5
4.1.6 Threads 5
4.1.7 Garbage Collector 6
4.1.8 Monitors 6
4.2 Event information in Java probe 6

5 Possibility to extend functionality **7**

5.1 Single Step event 7

6 Examples **7**

6.1 Garbage collector 7
6.2 Java Probes & Kernel Probes Mix 7

Appendix A **8**

Appendix B **10**

Appendix C **11**

1 Introduction

Inspired by the SUN DTrace ability to provide insight information about running java applications we think it makes sense to add this ability to the SystemTap as well. Basically we need a framework which allows investigation of the behavior of underlying operating system combined with the insight information about running Java applications. This can assist in identifying the underlying cause of a performance or functional problem. SystemTap simplify gathering of information about Linux kernel and theoretically it is possible to add support of the user applications. Current document describes our vision on how Java VM support could be integrated into the SystemTap.

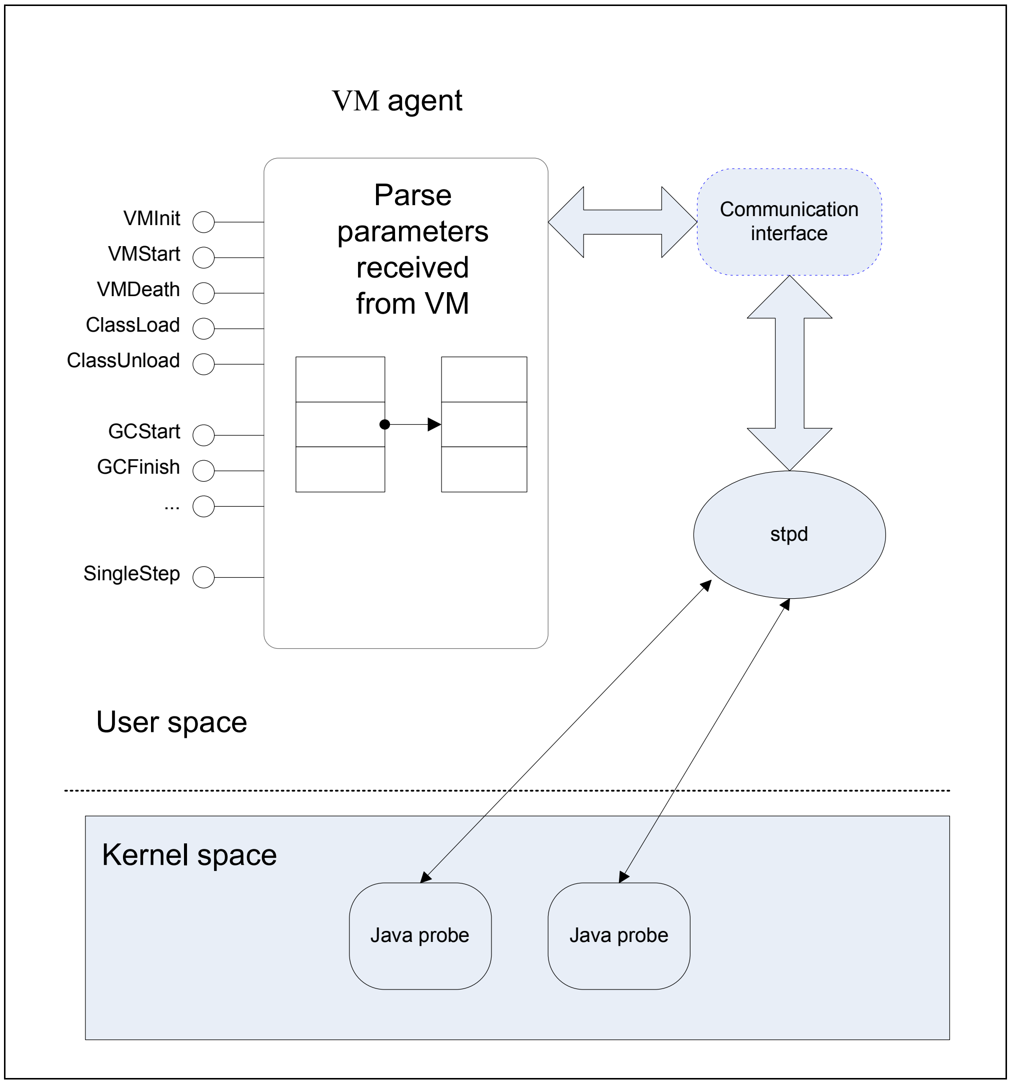
2 Java probes

We suggest adding a new Java probe into the SystemTap – it will be used to gather and process statistics for Java applications and Virtual machines. This probe corresponds to the current ideology of the SystemTap and is similar to the kernel probe for functions inside kernel. The difference is that we can't directly instrument a VM or Java application from SystemTap, but are able to do this with a cooperating VM agent. This way allows us to track VM activity as well as the Java application. Then we could make a direct communication between VM agent and Java probe using stpd daemon.

There are some JVM instrumentation interfaces: JVMTI, JVMPI, and JVMDI. Based on these interfaces one can create software agents to monitor Java VMs and applications. VM agent is a module that gets dynamically loaded into a VM at startup. The Java probes can be executed in the event callback functions of the VM agent.

In general the communication scheme could look like:

1. Initialization
   a. User executes Java application with VM agent
   b. Than user runs 'stap' for Java probe compilation and execution.
   c. Java probe sends message containing list of the interesting methods, classes, exceptions, etc to the VM agent.
2. Processing
   a. VM agent gets a control on a number of VM events (method enter, method exit, gc start, gc finish, etc)
   b. VM agent requests detailed information from VM using JNI/JVMTI/etc methods and passes it to the Java probe. This information is wrapped into the event dependent structures.
   c. Java probe gets a control after that and process information received from the VM agent.



Picture 1

Due to Java method naming the probe syntax is a bit more complex. Our suggestions about syntax are described in "Compiler" part.

3 VM – stpd communication interface (CI)

3.1 Data streaming interface

We propose to use well-defined data streaming interface to set up the communication between VM agent and stpd. There are a number of reasons that a data streaming interface is desirable for exposing Java probe data to SystemTap. One key reason is that to maximize the extensibility of Java probes, we wish to ensure that we can leverage existing instrumentation technologies provided by VM vendors and other open source projects. As an example, Eclipse/TPTP includes a sophisticated "probekit" technology designed to enable advanced byte code instrumentation. Leveraging this type of technology to create efficient Java Systemtap probes is desirable, (and necessary for advanced, extensible java probes). A data streaming interface avoids link issues and enables us to leverage these technologies.

3.2 Communication interface transport

CI could use any standard inter process communication protocol as a transport. There is a line of options, including pipes, memory mapping and sockets. Which one should be chosen? This is an open question for a moment. However, we know that the protocol should be streaming, fast, bi-directional, stable and well known. It should also provide a way for a stpd to connect to the VM agent on a latter stage, the "listening" end in this case should be unique for each agent instance. Let us assume that we're talking about unix domain sockets.

3.3 CI details

3.3.1 VM agent to stpd flow

VM agent sends various events received from VM side to stpd. Each event structure is started with the magic code to identify event type. The variable length data is encoded according to p. 3.3.3 and referred in tables as '*string*' and '*argument*'.

The names of methods, exceptions and classes are represented according to the JVMTI specification. For instance:

```
java.lang.System.arraycopy(Ljava/lang/Object;ILjava/lang/Object;II)V
Ljava/lang/Object
Ljava/lang/Object
```

Various IDs are being sent to the stpd side "as is" without any conversion and are provided only as a service information. Please note: there is no way to change any object/thread ID from the stpd side.

Please see the complete list of commands in the Appendix A.

3.3.2 stpd to VM agent flow

stpd sends service commands related to registered JavaProbes, SystemTap startup/shutdown, requests for additional data, etc to the VM agent. stpd uses the commands listed in the Appending B to specify methods/classes/exceptions of interest. They allow VM agent to drop most of the events from the VM and process only a few of them. This filtering helps to dramatically reduce the system overhead.

Please refer to the Appending B for the complete list of commands.

3.3.3 strings and optional data placement

Method/classes names, arguments and return values have variable length in the stream, so we need to provide the information about the length (or type) before the actual data to read this data successfully.

See Appendix C for details.

4 Compiler

4.1 Java probe syntax

SystemTap compiler should be extended to support Java probes. The fact that VM agent sends event-dependent information should also be taken into account. For instance the 'exception probe' should be able to get the information about the method which threw/caught an exception.

4.1.1 Java related structures

Every probe receives an event dependent structure described in the corresponding tapset. In fact this structure could be an array and the tapset is used to make it more readable. We are not going to describe these structures here because they depend on an event type as well as on a particular implementation of the VM agent. For instance the basic implementation assumes getting the line number information from `jlocation`, however we can also get the parameters and return values for the java-method-probe, stack trace for exception-probe, etc using BCI (see 4.2). We expect the event dependent structures to evolve as a wider range of Java events are exposed to SystemTap.

4.1.2 Methods

Java probe starts with the keyword probe followed by a description of where to place the probe and the script body to run when the probe executes. For instance, probe for `java.lang.System.arraycopy` method can be written as:

```
probe java.method("java.lang.System.arraycopy(Ljava/lang/Object;ILjava/lang/Object;II)V").enter
{
  …
  script body
  …
}
```

Therefore, common semantic looks like:

```
probe java.method("METHOD_SIG").EVENT_TYPE
```

Where:

METHOD_SIG	Signature of the method. It includes fully qualified method name and allows an asterisk “*”, which means that resulting probe should care about all methods with matched names.
EVENT_TYPE	The type of event to catch. It can be `enter/return/compile`. This list could grow in the future according to VM agent abilities. This part is optional, by default assumes value `enter`.

4.1.3 Classes

The `java.class` probe is useful to track class-loading process.

```
probe java.class("CLASS_NAME").EVENT_TYPE
```

CLASS_NAME	Fully qualified class name. Wildcard is allowed as well as in METHOD_SIG
EVENT_TYPE	`load` - class is first loaded
	`prepare` - class preparation is complete. Class fields, methods, and implemented interfaces are available, and no code from the class has been executed
	unload – class is unloaded by VM

4.1.4 Exceptions

The `java.exception` probe could be used to trace exception-related activity of Java application.

```
probe java.exception("EXCEPTION_NAME").EVENT_TYPE
```

EXCEPTION_NAME	Fully qualified exception name. Wildcard is allowed
EVENT_TYPE	`thrown` – exception is thrown
	`catch` – a thrown exception is caught

4.1.5 VM events

These probes could be used to trace VM activity.

```
probe java.vm.EVENT_TYPE
```

EVENT_TYPE	`init` – VM initialization is completed
	`death` – VM is terminated

4.1.6 Threads

The `java.thread` probes could be used to trace thread events.

```
probe java.thread.EVENT_TYPE
```

EVENT_TYPE	`start` – thread started, no initial methods executed `end` – thread is finished

4.1.7 Garbage Collector

The `java.gc` probe could be used to track garbage collector activity.

```
probe java.gc.EVENT_TYPE
```

EVENT_TYPE	`start` – Garbage collector begins full cycle `finish` – Full cycle is finished

4.1.8 Monitors

The `java.monitor` probe could be used to track Java programming language monitors.

```
probe java.monitor.EVENT_TYPE
```

EVENT_TYPE	`contended_enter –` thread is attempting to enter a Java programming language monitor already acquired by another thread `contended_entered –` thread enters a Java programming language monitor after waiting for it to be released by another thread `wait –` thread is about to wait on an object `waited –` thread finishes waiting on an object

While this details an initial set of events, there is significant opportunity for growth. Heap allocation & collection events, finalization events, object tags, etc… Note that one may also want to have probe execution make additional requests over the data interface.

4.2 Event information in Java probe

According to SystemTap ideology every probe should check method (exception, field, etc) name before execution. For instance:

```
probe java.method("java.lang.System.arraycopy(Ljava/lang/Object;ILjava/lang/Object;II)V")
```

This probe should check if full method signature equals to the specified (`java.lang.arraycopy…`). So there are three ways:

1. VM agent catches `JVMTI_EVENT_METHOD_ENTRY` event and executes every java probe. Than every probes make this check itself.
2. VM agent catches `JVMTI_EVENT_METHOD_ENTRY` event and tries to find method name in the list sent by java probes. If it's found then VM agent calls corresponding probe.
3. VM agent doesn't catch `JVMTI_EVENT_METHOD_ENTRY` but instruments interesting methods using BCI (Bytecode Instrumentation). In this case the list of these methods also required for VM agent.

The first and the second ways use JVMTI events to start method entry probe, and could be used for simple tracking of which method were called during application execution – they do not provide an ability to track parameter values passed to the method etc. These ways could be used in a very simple java probes were having such information is unnecessary. Both the first and the second ways require additional events filtering either on the probe or on the agent side. The first way is also considerably slow and puts additional load on I/O interfaces between the agent and a probe and is less applicable to our needs.

The second way is easier than the third one, but bytecode instrumentation (BCI) is a most flexible way to deal with method events as it makes parameter extraction as well as other more complex analysis possible.

Note that some JVMs choose not to implement JVMTI_EVENT_METHOD_ENTRY. They do this because the preferred method to extract this type of information is using BCI. This places additional overhead on the VM Agent developer as BCI is more complex than using basic JVMTI_EVENTs. For ***advanced***, portable, agents that perform BCI, leveraging existing, open source BCI interfaces such as those within Eclipse/TPTP is desirable. This is because reinventing a sophisticated BCI framework within SystemTap would be inefficient. A VM agent leveraging Eclipse BCI technology can expose Java probe information to Java probes to SystemTap stpd via the communication interface.

Both ways are based on information received by VM agent from java probe. Java probe should send methods-to-be-probed list to the VM agent at the Initialization phase (as described in "Java probes" part). In our example the probe should send to the VM agent the following information:

```
java.lang.System.arraycopy(Ljava/lang/Object;ILjava/lang/Object;II)V
```

5 Possibility to extend functionality

5.1 Single Step event

It is possible to use Single Step callback provided by JVMTI specification to extend number of events. A single step event is generated whenever a thread reaches a new location. Typically, single step events represent the completion of one VM instruction. For example it's possible to implement custom event handler like 'executed bytecode aastore' or so on. In this case JVMTI agent should analyze current state of the VM on every call and execute suitable probe. Unfortunately it's very SLOW. The second issue – it's a VM-dependent feature, because some implementations may define locations differently. Anyway even if VM doesn't support Single Step feature at all then corresponding probes just will not be executed. So we can consider this feature as 'optional'.

6 Examples

6.1 Garbage collector

The following example demonstrates using Java probes to measure GC pause time and total GC time:

```
probe java.gc.start {
       t = gettimeofday_ms()
}

probe java.gc.finish {
       printf("time = %d\n", gettimeofday_ms() - t)
}

probe begin {
       tt = gettimeofday_ms()
}

probe end {
       printf("total time = %d\n", gettimeofday_ms() - tt)
}

global t, tt
```

6.2 Java Probes & Kernel Probes Mix

Imagine that we should investigate behavior of the `java.io.BufferedWriter` class.

We will use the following test written in Java:

```
// test.java

import java.io.*;

public class test {
    public static void main(String[] args) {
        try {
            FileWriter caw = new FileWriter("test.tmp");
            BufferedWriter bos = new BufferedWriter(caw);
            for (int i = 0; i < 14000; i++)
                bos.write('a');
            bos.close();

            caw = new FileWriter("test.tmp");
            bos = new BufferedWriter(caw, 4096);
            for (int i = 0; i < 14000; i++)
                bos.write('x');
            bos.close();

        } catch (IOException e) {
            System.err.println("Unexpected IOException: " + e);
        }
    }
}
```

For debugging system calls we'll use kernel probe:

```
probe syscall.write {
        if (execname() == "java")
                printf("  %s\n", argstr)
}
```

For profiling of the `write` method we'll use the following Java probe:

```
probe java.method("java.io.BufferedWriter.write(Ljava/lang/String;)V").enter {
       printf("--- BufferedWriter.write.enter ---")
       t = gettimeofday_ms()
}

probe java.method("java.io.BufferedWriter.write(Ljava/lang/String;)V").return {
       printf("time = %d\n", gettimeofday_ms() - t)
       printf("--- BufferedWriter.write.return ---")
}

probe begin {
       tt = gettimeofday_ms()
}

probe end {
       printf("total time = %d\n", gettimeofday_ms() - tt)
}

global t, tt
```

An output allows us to evaluate an efficacy of the buffering. For instance "gij (GNU libgcj) version 4.0.0 20050519" always calls `write` with buffer size equals to 250 bytes and it doesn't correlate with the buffer size inside the `BufferedWriter` class.

Appendix A

Note: `jthread`, `jobject` are types defined in JVMTI/JNI specification.

`MethodEnter` event

Name	Length	Comment
Event code	1 (unsigned char)	0x80
Method name	variable (string)	
Thread ID	sizeof(jthread)	
Number of optional records	1 (unsigned char)	

Optional record (if any)	Variable	Optional data including method parameters in case if agent has optional ability to pass them to the probe. See Appendix C for details about optional data placement.

`MethodExit` event

Name	**Length**	**Comment**
Event code	1 (unsigned char)	0x81
Method name	variable (string)	
Thread ID	sizeof(jthread)	
Popped by exception	1 (bool)	
Return value	variable (argument)	See Appendix C for details about optional data placement.

`MethodCompile` event

Name	**Length**	**Comment**
Event code	1 (unsigned char)	0x82
Method name	variable (string)	
Code size	4 (unsigned long)	
Optional data	Variable	

`ClassLoad` event

Name	**Length**	**Comment**
Event code	1 (unsigned char)	0x83
Class name	variable (string)	
Thread ID	4 (unsigned long)	

`ClassPrepare` event

Name	**Length**	**Comment**
Event code	1 (unsigned char)	0x84
Class name	variable (string)	
Thread ID	sizeof(jthread)	

`ExceptionThrown` event

Name	**Length**	**Comment**
Event code	1 (unsigned char)	0x85
Exception name	variable (string)	
Thread ID	sizeof(jthread)	
Method name	variable (string)	

`ExceptionCatch` event

Name	**Length**	**Comment**
Event code	1 (unsigned char)	0x86
Exception name	variable (string)	
Thread ID	sizeof(jthread)	
Method name	variable (string)	

`ThreadStarted` event

Name	Length	Comment
Event code	1 (unsigned char)	0x87
Thread ID	sizeof(jthread)	

`ThreadFinished` event

Name	Length	Comment
Event code	1 (unsigned char)	0x88
Thread ID	sizeof(jthread)	

`MonitorEnter` event

Name	Length	Comment
Event code	1 (unsigned char)	0x89
Thread ID	sizeof(jthread)	
Object ID	sizeof(jobject)	Object ID passed to the probe only for tracking purposes. For instance it is allow us to understand how often object with specified ID used in synchronized blocks.

`MonitorEntered` event

Name	Length	Comment
Event code	1 (unsigned char)	0x8A
Thread ID	sizeof(jthread)	
Object ID	sizeof(jobject)	

`MonitorWait` event

Name	Length	Comment
Event code	1 (unsigned char)	0x8B
Thread ID	sizeof(jthread)	
Object ID	sizeof(jobject)	
Timeout	4 (unsigned long)	

`MonitorWaited` event

Name	Length	Comment
Event code	1 (unsigned char)	0x8C
Thread ID	sizeof(jthread)	
Object ID	sizeof(jobject)	
Timeout	4 (unsigned long)	

Appendix B

`RegisterMethod` command

Name	Length	Comment
Command code	1 (unsigned char)	0x40
Method name/mask	variable (string)	

`RegisterClass` command

Name	Length	Comment
Command code Class name/mask	1 (unsigned char) variable (string)	0x41

`RegisterException` command

Name	Length	Comment
Command code Exception name/mask	1 (unsigned char) variable (string)	0x42

`GCEvents` command

Name	Length	Comment
Command code Enable	1 (unsigned char) bool	0x43

`ThreadEvents` command

Name	Length	Comment
Command code Enable	1 (unsigned char) bool	0x44

`MonitorEvent` command

Name	Length	Comment
Command code Enable	1 (unsigned char) bool	0x45

`RequestForOptionalData` command

Name	Length	Comment
Command code Options mask	1 (unsigned char) 4 (unsigned long)	0x46

Appendix C

Strings placed into the stream as follow:

Name	Length	Comment
Length Data	4 (unsigned long) specified by length field	

Optional data (methods arguments, return value, etc.):

Name	Length	Comment
Type Data	1 (unsigned char) depends of type	