Integrated Server for Dynamic Program Analysis

Vít Kabele
vit@kabele.me

Context

Software is nowadays a part of our everyday lives. As software drives even the most dependable task in our world, the requirements on code quality are higher than ever before. Current applications are, however, extremely complicated and their lifecycle often involves more than a dozen of programmers, code reviewers or testers. With such a complexity, no one is able to keep all project details in his mind and searching for a bug or fixing a performance issue becomes Herculean tasks.
In these conditions, developers are looking for more advanced approaches to gain deeper insight into their products. One such approach is called Dynamic Analysis.

Dynamic Analysis

Dynamic analysis is a way of observing the properties of a running program. Curious programmers can use this approach to reveal the hot-paths in a program to perform further optimization on them. Others might be interested in memory usage by analysing the behaviour of the garbage collector. A nifty hacker can use it to perform reverse engineering. All of these tasks can be done using one of the existing Dynamic Analysis tools. The difference is in the difficulty of doing so and in the accuracy. The ShadowVM tool was created with the aim to improve accuracy and simplify the usage.
Although the available tools might seem different on the first sight, they share a few common concepts under the hood. The one used in DiSL is called bytecode instrumentation. DiSL is created for the purpose of analysing instances of the Java Virtual Machine.

Bytecode Instrumentation

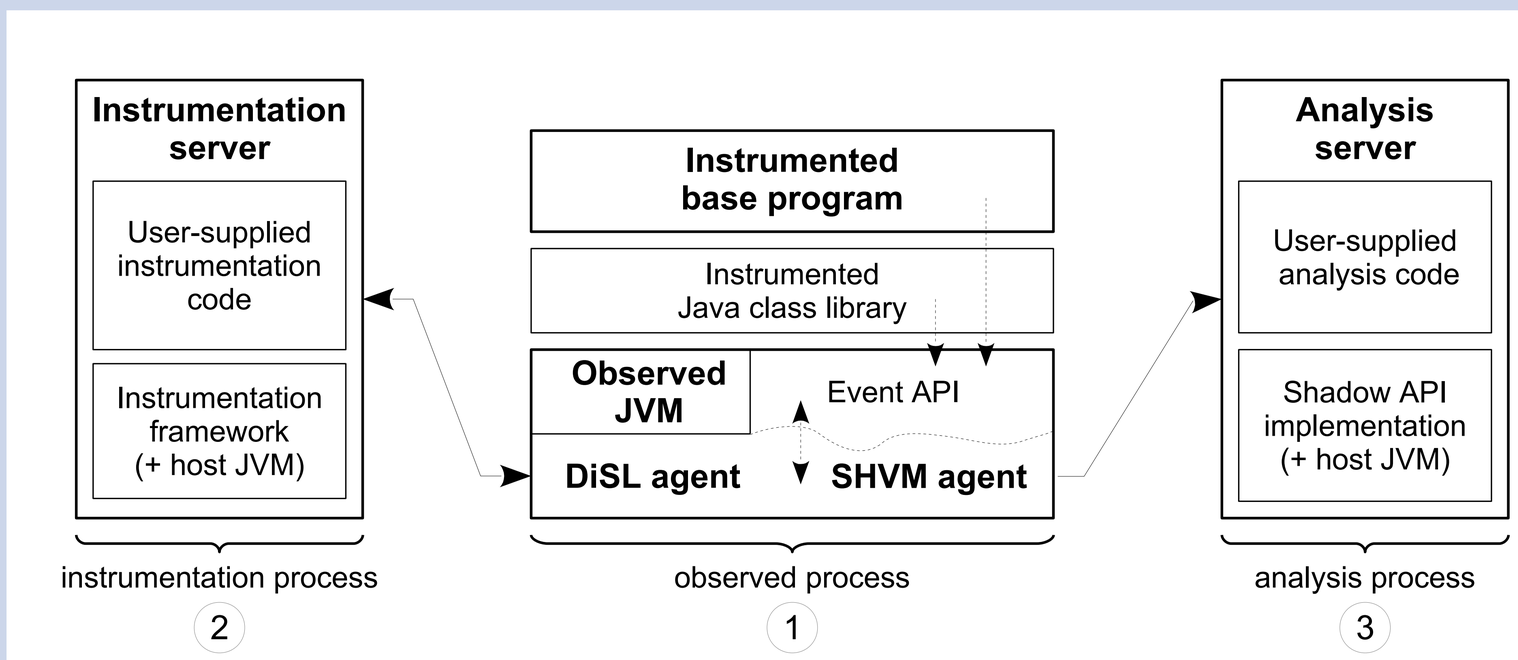
A Bytecode Instrumentation is based on inserting predefined pieces of bytecode into predefined places in the original program. Although this approach can be used to alternate the code execution in almost any way, the dynamic analysis usage is limited to only add checkpoints into the program control flow path.

Problem

The problem with both the DiSL and the ShadowVM is that in some real heavyweight analysis they can affect performance of the observed application in a non-trivial way. This is caused mainly by the need of transfering all the application classes to remote servers. Our work is targeting this issue by alternating the architecture of the whole suite to better reflect the requirements and reducing the amount of data sent over the network.

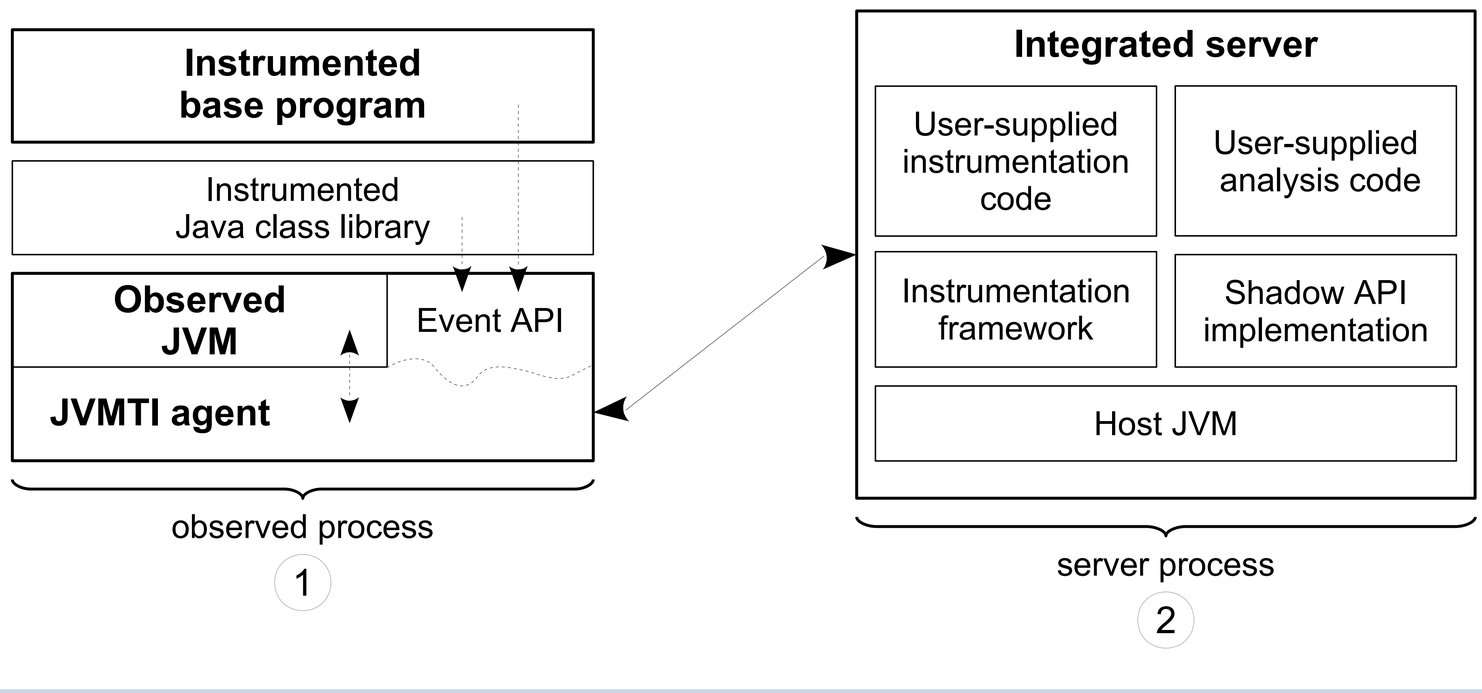
Former architecture

Formerly, the suite was composed of four main basic blocks. The **DiSL client**, responsible for catching the loaded classes in the client Virtual Machine. These classes were sent to the **DiSL server** where the instrumentation was performed and the instrumented classes were sent back to the client. The client passed the modified bytecodes back to the JVM and the class loading process continued.
Along with the DiSL client, the **ShadowVM client** was loaded to the observed Virtual Machine. This client listened for analysis events (mainly) triggered from the inserted instrumentation code. These events were collected in a queue and then sent to the **ShadowVM server**, also known as Analysis server.
Both servers are written in the Java language. This allows them to take advantage of builtin features like Reflection. The clients are written in the C programming language and they use the JVMTI/JNI interfaces of the Java Virtual Machine. This arrangement is shown in the diagram.



New architecture

The main difference between the old and the new architecture is that both servers are now merged into one executable. This was crucial in order to reduce the amount of transferred data, because the backends in this arrangement are able to share the reflection information. The class loading related network traffic on the client is therefore about 50% smaller than before.
As a consequence of such a decision it turned out that the native agents needs also needs to be merged. This necessity has risen from the fact, that the server requires proper ordering of messages, which was not possible to achieve while keeping the clients separated. By merging the clients we removed many of the duplicate code and by this we reduced the total amount of C code. This lead to better maintainability of the whole solution.
The new arrangement of the merged client and merged server allowed us to further optimize the network communication as it is now performed over just one socket per one client. It includes designing a whole new network protocol that combines both instrumentation and analysis related messages and their respoctive responses. This new protocol is build using the Google Protocol Buffers library[a].
Finally, we implemented session support into both client and server part of the suite. This decouples the observed Virtual Machine from the server and therefore allows the analysis developer to run a single instance of DiSL/ShadowVM server in the background and it is not required to reconfigure the instrumentation manually for every analysed JVM instance.



[a] https://developers.google.com/protocol-buffers/