Contemplate
THINKING ABOUT SOFTWARE

Case Study: ThreadSafe Identifies Apache MINA Bug

Executive Summary

Contemplate is developing next-generation static analysis tools for finding concurrency problems. As part of our testing, we have been analysing open source projects that use concurrency. One project that we have used for testing is Apache MINA.

Our **ThreadSafe** tool identified an issue related to concurrency in part of the framework that can be used for building a TCP server based on the Java non-blocking I/O libraries (java.nio). The way in which a Java library was used could lead to undefined behaviour, potentially including an infinite loop, crash or resource leak.

The issue was reported to the developers, who committed to fixing the bug in the next release. As the issue was detected in a pre-release snapshot and is in recently added code, it has been caught before making it into a release version.

Apache MINA

Apache MINA is a ‘network application framework’ intended to support the development of high-performance, scalable network applications. There are several projects hosted by the Apache Software Foundation that use Apache MINA, including clients and servers for HTTP, FTP, SSH, and XMPP. Apache MINA is also used in QuickFix/J: an open-source implementation of the FIX (Financial Information Exchange) protocol.

The Problem

The issue identified is in a snapshot version of code prior to the release of 3.0.0 M1. The class containing the issue did not exist in previous versions of MINA. Thus the issue was identified and could be corrected before the relevant code was released.

Contemplate **ThreadSafe** found the issue in a class called `NioTcpServer`, which is part of the MINA framework used for building TCP servers based on the Java non-blocking I/O libraries (java.nio). The class contains a field called ‘`addresses`’ which is used to track which sockets the TCP server is listening on. It is defined as follows:

```
// list of bound addresses
private Set addresses = Collections.synchronizedSet(new HashSet());
```

The type of the field is `java.util.Set<java.net.SocketAddress>`. The field is assigned when the server class is instantiated by creating a `java.util.HashSet` instance and ‘wrapping’ the result using `Collections.synchronizedSet`. This ensures that all accesses to the Set are synchronized.

A `bind(SocketAddress...)` method is defined which binds TCP ports for the given sockets, and adds each bound socket to the collection described above. Each access to the set is synchronized because of the use of `synchronizedSet`.

The class also defines a method that can be used to unbind all currently bound ports, called `unbindAll()`. The method is defined as follows:

```
@Override
 public void unbindAll() throws IOException {
     for (SocketAddress socketAddress : addresses) {
         unbind(socketAddress);
     }
 }
```

This code is intended to iterate over the collection, and unbind any and all currently bound TCP sockets. However, the iteration over the set is not synchronized. Hence, if a socket is bound and added to the collection during the iteration, a race condition can occur. The most obvious consequence of such a race condition is a resource leak caused by sockets being left open. However, race conditions of this nature can have more serious consequences, which are discussed later in the *Conclusions* section.

The API documentation for `Collections.synchronizedSet` indicates that collections wrapped using this method must be synchronized when iterating over their contents. The behaviour of iterating over such a collection without synchronization is left undefined:

> `public static <T> Set<T>` **`synchronizedSet`**`(Set<T> s)`
>
> Returns a synchronized (thread-safe) set backed by the specified set. In order to guarantee serial access, it is critical that **all** access to the backing set is accomplished through the returned set.
> It is imperative that the user manually synchronize on the returned set when iterating over it:
>
> ```
> Set s = Collections.synchronizedSet(new HashSet());
>     ...
> synchronized(s) {
>     Iterator i = s.iterator(); // Must be in the synchronized block
>     while (i.hasNext())
>         foo(i.next());
> }
> ```
>
> Failure to follow this advice may result in non-deterministic behavior.

It is clear from the documentation that the collection must be synchronized when iterating over it. The `unbindAll()` method does not perform this synchronization, and hence, its use could result in

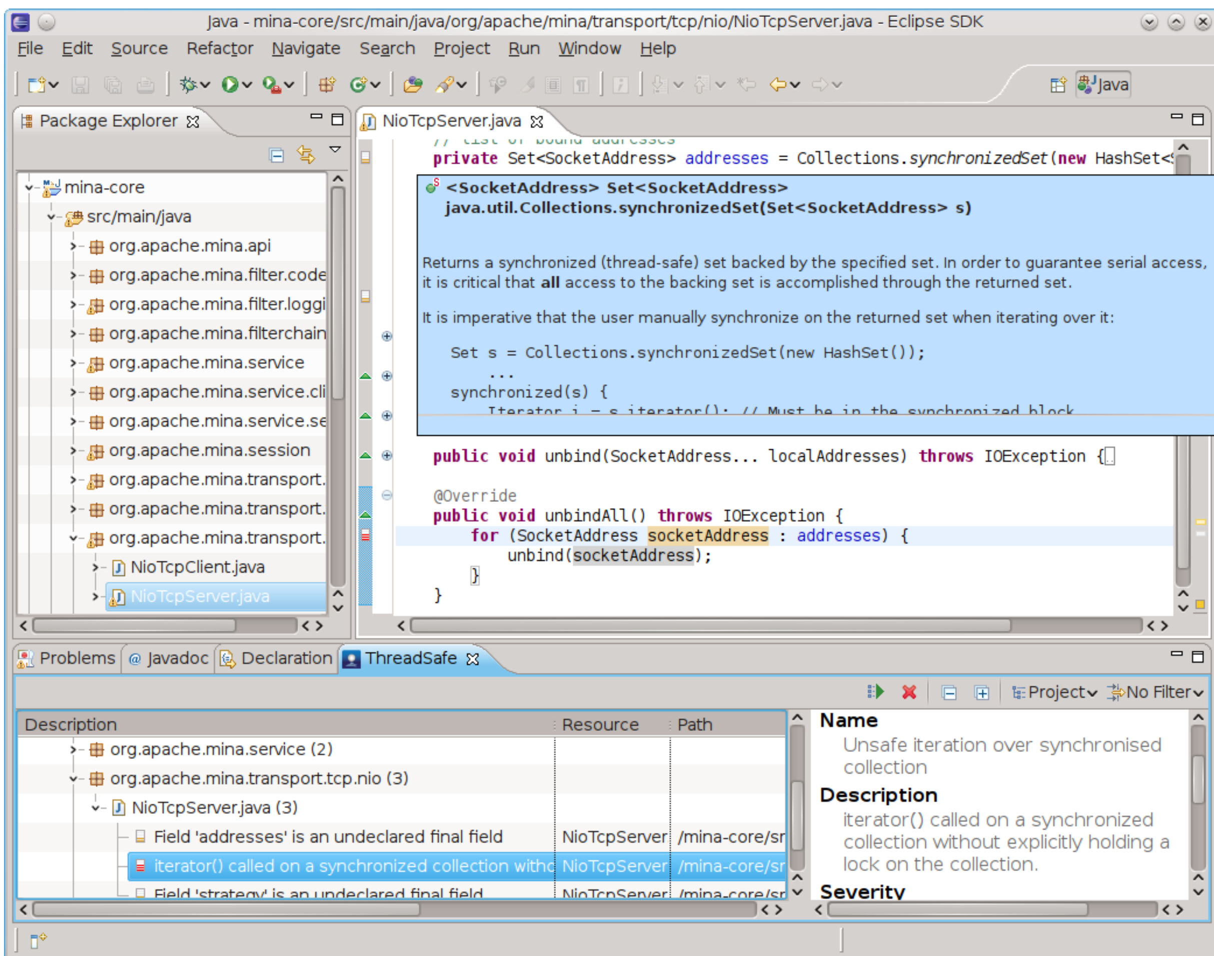
undefined behaviour. It is worth noting that as the behaviour is undefined, it may vary between different versions of Java, and between different platforms and architectures. It may also vary between different invocations of the method.

ThreadSafe

We analysed the MINA Core project by importing the project into the Eclipse IDE, and analysing it using the latest version of **ThreadSafe**, a product developed by Contemplate.

A screenshot showing the MINA code imported into Eclipse is included below. The class containing the bug is open in the IDE, and the screenshot shows the javadoc (copied above) that appears when the mouse is hovered over the call to `synchronizedSet`.

The screenshot also shows some of the findings of the ThreadSafe analysis. The currently selected finding (highlighted in blue) identifies the issue described above. A brief description of the issue can be seen in the bottom-right corner of the screen. Threadsafe also offers users a more detailed explanation of findings within the IDE, but this is not shown here.



Result

Contemplate reported the bug as an issue in the JIRA issue tracker for the project. [1]

The project developers responded shortly after we filed in the issue tracker. One active developer (who appears to have made over 340 commits to the project) commented:

> *"Good catch! I will see if I can fix the code asap. Thanks for the report!"*

The project lead then updated the 'Fix Version' for the issue, indicating that the issue should be fixed for the next release.

Conclusion

The issue we found and reported was considered important enough that a fix was scheduled for the next release, and the developers' comments suggest that they thought it unlikely that it would have been noticed otherwise.

Without a fix, a call to `unbindAll()` would have resulted in non-deterministic behaviour when a socket is bound and added to the addresses collection concurrently. Thus, it seems most likely to manifest in an application that opens and closes ports dynamically.

One of the projects built on MINA is an FTP server. The FTP protocol opens and closes connections for data transfer dynamically when used in passive mode (that is, the server opens a new port for a specific client, and sends the port number to the client). In this case, the set of addresses would not be static, and a call to `unbindAll()` (which could conceivably be called at server shutdown) could result in a race condition.

The non-determinism may mean that the new socket is bound or unbound after the call to `unbindAll()`. If the latter, then a socket resource might be 'leaked'. However, the issue could have more serious consequences. Concurrent access to collections without proper synchronization has undefined behaviour, and in some cases, this can lead to the code hanging in an infinite loop. [2]

---

[1] https://issues.apache.org/jira/browse/DIRMINA-869
[2] http://mailinator.blogspot.com/2009/06/beautiful-race-condition.html