Graded lab assignment: Chat server

1 Overview

In this exercise, you will analyze a chat server. The server uses the main thread to accept requests, and worker threads to handle incoming traffic from a client (see Figure 1). Each client (session) gets its own worker thread on the server side. To be able to run the exercise in JPF, we use *jpf-nas* to handle the client/server pair. In the assignments below, "correct behavior" means no uncaught exceptions, assertion violations, data races, or deadlocks.

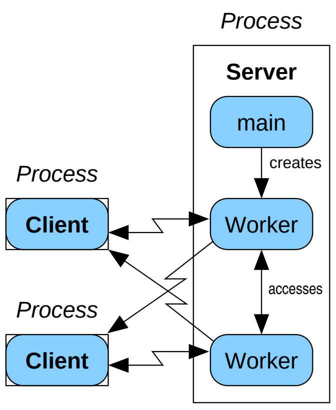

Figure 1: Overview of the chat server.

Coverage of certain system configurations can be shown as partial JPF executions (with logged screen output) or by temporarily adding `assert(false)` at the location of interest, and showing an error trace leading to that assertion; see Figure 2. If you print or log something inside the exception handler, you can inspect the execution output to confirm that this statement has been reached at least once. To get an execution trace to the location in question, `assert(false)` will entice JPF to produce a counterexample leading to that statement.

```
try {
    ...
} catch(IOException ioe) {
    System.err.println("Worker thread " + idx + ": " + ioe);
    // assert(false); // uncomment to show that this handler is used
}
```

Figure 2: How to show that an exception handler is tested.

2 Assignments: Mandatory tasks

2.1 Installation of jpf-nas

The extension jpf-nas is available from the same GitHub organization as JPF: you have to use `ant` to compile the sources:

```
cd ~/jpf # assuming jpf-core is in ~/jpf/jpf-core
hg clone https://github.com/javapathfinder/jpf-nas
cd jpf-nas
ant
```

You also have to enable the extension `jpf-nas` in `~/.jpf/site.properties`. You can either add two lines:
`jpf-nas = ${jpf-home}/jpf-nas`
after the declaration of jpf-core, and

`extensions+=,${jpf-nas}`
at the end, or use this script (with `bash`):

```
[ -e ~/.jpf ] || mkdir ~/.jpf
mv ~/.jpf/site.properties{,old} # make a backup
cat > ~/.jpf/site.properties << 'EOF'
jpf-home = ${user.home}/jpf
jpf-core = ${jpf-home}/jpf-core
jpf-nas = ${jpf-home}/jpf-nas
extensions=${jpf-core}
extensions+=,${jpf-nas}
EOF
```

2.2 Diagnosis

The extension jpf-nas automatically simulates multiple processes inside JPF. It also includes *finalizer threads,* which run when the *main* thread of a process terminates. The finalizer threads do not affect the outcome of the program, so you can ignore them (threads 3–5).

Thread	0	1	2	3–5	6	7
Trans.	Server main	Client 1	Client 2	Finalizers	Worker 1	Worker 2
0	init, call accept					
1		init, connect				
2	finish accept, start worker					
3–4	call accept					
5		write, call readLine				
6			init, connect			
7	accept					
8	start worker					
9	close port					
10	shut down					
11–12				[no source]		
13–18					register worker	
19–21					readLine	
22–29					sendAll	
30–31		readLine, close socket				
32				[no source]		

Table 1: Example showing how thread transitions can be summarized.

1. Diagnose the error trace shown by JPF. Use a table like Table 1. If you have many transitions with "ins[truction]s w[ith]o[ut] sources", you can summarize these as "[no source]", perhaps together with more interesting actions of other threads, such as shown in column 6 of the partial trace in Table 1.

2. Give an explanation of the error trace. Show how the server and clients interact, and how the failure happens. This can be done as an enumeration of about 5–12 key actions, with a short phrase explaining their meaning and impact on the program's behavior.
   Note: a first summary as shown above may still include a lot of detail that is not needed to understand the root cause of the trace. Once you have finished summarizing the trace, look again at the exception itself and where it occurs. Then, focus on field accesses that (presumably) caused the exception.

3. Fix the defect; provide a patch. After the fix, JPF won't terminate for a long time, so you can stop it after a couple of seconds.

3 Optional tasks

3.1 Validation, documentation

1. Explain why your solution is likely correct. Are workers added (registered) and removed in a thread-safe way? Is the access to sockets of other workers always safe (no null pointer dereference; possible exceptions handled)?
2. What can you say about the resulting system after it has been verified with JPF? Is it proven to be correct? Why (not)?

3.2 Resource initialization

1. Initialize the communication resources `in` and `out` of the worker threads in the constructor of `Worker`. In that constructor, throw a `java.util.IOException` if a resource cannot be initialized. Handle the exception in the `ChatServer` and do *not* launch a new thread if the constructor failed. (In this case, you cannot send a message back to the client because communication is not available.) Note that this change may affect how other attributes (not related to resource initializations) are set, in the constructor or in the `run` method.
2. Simulate a case where some I/O resource cannot be initialized by injecting an exception (directly in the server code) using `gov.nasa.jpf.vm.Verify.getBoolean`. Make sure your program handles the resulting exception correctly. Show a partial model checking execution that gives evidence of successful fault injection. The best way is to temporarily add `assert(false)` at a place where you want to prove that the location can be reached.

3.3 Thread handling

The given implementation launches a new thread for each connection. Connections are accepted no matter what the current server load is. A smart server has a limit on the number of *currently active* threads, and perhaps also ways to "recycle" worker threads.

Library class `java.util.concurrent.ThreadPoolExecutor` provides all the necessary functionality, together with classes `BlockingQueue`, `RejectedExecutionHandler`, `SynchronousQueue`, and `TimeUnit` from the same package. See:
`https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html`.

1. Change your implementation to use `ThreadPoolExecutor` to launch the worker threads. This may require adaptions in the way I/O resources are managed (see Section 3.2). Ensure your new implementation works correctly.
2. The real implementation of `ThreadPoolExecutor` is rather complex and uses a background thread to manage the tasks. It cannot be checked by JPF version 8. Replace the real class with a *model class* that implements the following:
   (a) A compatible constructor.
   (b) An `execute` method that keeps track of the number of threads launched, launches a thread if the maximum number has not been reached yet, and otherwise rejects the request by calling the `RejectedExecutionHandler`. Note: No need to keep track of the number of *active* threads precisely in the model class; just use the number of launched threads as an approximation (upper bound). Also the rejected case can be handled directly, *without* launching a new thread for this, to keep the state space smaller.
   (c) An empty stub for `shutdown`.
3. Test settings where all (two) threads are accepted or only some (0 or 1) of them: Show how you configure the system (parameters used for the ThreadPoolExecutor) to simulate these cases. Show a partial JPF execution trace that gives evidence of the `RejectedExecutionHandler` being executed under the right settings (limitation).

Note: The archive of the exercise provides code skeletons for the two classes that are needed; you need only modify `env/java/util/concurrent/ThreadPoolExecutor.java`.

3.4 Client-side verification

Write properties in `ChatClient` to ensure that received message is always one of:

1. One of the messages sent by clients.
2. “Server busy”.

Show the resulting property violation (just the property, not the full trace); amend the property by taking into account all cases.

3.5 Additional properties

1. Can you come up with an additional meaningful (non-trivial) property? Write that property as an assertion.
2. How do you check if the different combinations of clients, worker threads serving clients, and messages between clients actually occur? Can you check this as an assertion? Hint: look at the log/console output.

3.6 Bonus

If you find other defects in the example, or even in Java Pathfinder, and can create a small test to reproduce it, you can get an extra 0.25 points, at the discretion of the examiner. A fix of said bug can give an additional +0.25 points. The total number of such extra points cannot exceed 0.5 points.

4 Grading criteria

Description	Code	Trace	Doc.	Points
There is a coherent and convincing argument on the correctness of the resulting system design.			✓	+0.5
There is a correct assessment on the soundness of the verification methodology.			✓	+0.25
All initializations that may fail, happen in the constructor of `Worker`; exceptions are correctly handled by the caller.	✓			+0.5
Exceptions are successfully simulated and handled.	✓	✓		+0.5
`ThreadPoolExecutor` is correctly used.	✓			+0.5
`ThreadPoolExecutor` is correctly modeled.	✓			+0.5
Executions of rejected executor requests are covered (shown as partial execution or error trace).		✓		+0.25
Client-side properties are implemented. An error trace shows the initial property violation, and the final property is correct.	✓	✓		+0.5
A meaningful additional assertion is added.	✓			+0.25
A correct argument is presented on how to check the occurrence of permutations of messages showing clients and worker threads.			✓	+0.25

Table 2: Extra points for advanced features/properties. “Doc.” represents documentation beyond short comments in code, which are subsumed by “Code”.

Pass (E):

- The submitted work is your group’s own original work.
- The mandatory grading criteria (see Section 2) are fulfilled.

Extra points are awarded according to Table 2.

Reduced points for late submissions! The maximal number of points is reduced by 1 for each missed deadline. There are two deadlines: for the small lab exercises (tutorial) and for this exercise.