Improvements for the Simulated Car Racing Software Interface

Jan Quadflieg

Abstract—We analyze the timeout behavior of the server bot used in the simulated car racing championship and identify two shortcomings. Using Windows, we observe that the implementation generates timeouts too early. On both Windows and Linux, packets arriving too late introduce a lag from which a car controller can never recover. We present a solution for both problems in form of a modified implementation which adds a gametick counter to the network protocol and uses polling to implement the timeout behavior in a reliable way.

I. INTRODUCTION

In the last years, the Simulated Car Racing Championship (SCRC) offered participants the possibility to evaluate artificial and computational intelligence methods in the context of a racing game. Using TORCS, the open racing car simulator, the organizers of the SCRC provided a stable and well defined interface based on text messages which are sent over a UDP connection. The bot provided by the organizers integrates into the game, gathers the necessary information (position on the track, speed of the car, etc) and sends a packet containing all the data to the client. The race runs in real-time, which means that the client has only 10 ms to process the incoming sensordata and send its reply. If a controller has not replied after 10 ms the server bot reuses the action from the last gametick. It is up to the controller to avoid the possible timeouts.

This work was motivated by experiences gained while doing the experiments for our contribution to a special issue of Genetic Programming and Evolvable Machines [2]. To compare several controllers with our own SCRC controller Mr. Racer we evaluated them ourselves, in contrast to simply taking results from the literature. During these evaluations, some of the controllers generated timeouts, which in general changes the results for the worse. If that happened, we repeated the evaluation to treat the other controllers as fair as possible but no matter how hard we tried, we could not get rid of all timeouts. Sometimes we even had the impression that the controllers run fast enough and that the game generates unnecessary timeouts.

The rest of this work is structured as follows: We analyze the behavior and identify shortcomings of the implementation in section III. Our suggested improvements are presented in section III. We close with summary and conclusions.

II. CURRENT IMPLEMENTATION AND OBSERVED BEHAVIOR

The current implementation of the SCRC server uses the select function to check if the socket is ready to read, i.e. if the answer of the controller has arrived. The timeout parameter of the function offers the possibility to only wait a certain amount of time, which is used in the SCRC server to implement the timeout - a simple and straight forward solution. The select function is part of the old Berkeley socket API and is available on both Windows and Linux. But using the Windows binaries provided by the organizers, we always had the impression that the server sometimes generates timeouts although the controller answers fast enough. This is especially true when using the textonly mode, which would be ideal for experiments (offline optimization, etc) but always generates so much timeouts that it is practically impossible to use. And it was always unclear to us how the server deals with the arriving packets after a timeout occurred. Therefore, we want to measure how long the server actually waits before a timeout occurs and observe how the server handles packets which arrive too late.

Pre-experimental planning To test the behavior of the server implementation, we isolated the network part of the SCRC bot and built a standalone version independent of TORCS, which can be compiled on machines running Windows and Linux. This dummy server does the same handshake procedure to identify a client as the SCRC server and then simulates a number of gameticks by sending a message to the client and waiting for the response. All this is done using the original network code. To gather data, we added the following functionality: The message send to the client contains the gametick counter and the time spend calling the select function is measured using a high-resolution timer. On Windows the latter is done using QueryPerformanceFrequency and QueryPerformanceCounter according to the official documentation [3], on Linux we use clock_gettime with the CLOCK_REALTIME parameter. Using the high resolution timers, we also control the time waited between two consecutive gameticks (using a loop, instead of e.g. sleep). This gives us the possibility to simulate the behavior of the game, with e.g. 20 ms between packets (TORCS running with graphics) and little or no waiting between packets (TORCS running in textonly mode).

As a client, we use a modified version of the Java client. The client simply answers by returning the original message, which contains the gametick counter. This gives us the possibility to observe at which time the server processes which answer. The client also uses a high resolution timer (System.nanoTime) and a loop to wait a certain amount of time before sending the answer. By doing this, we can simulate a controller which needs e.g. 4 ms to compute its response and it also gives us the possibility to cause timeouts on purpose by waiting too long. Note that busy waiting,
although generally undesirable, is the only way to somehow reliably wait for the small amounts of time used here.

**Task** Verify that the server indeed waits for 10 ms before a timeout is generated. Observe the handling of late packets.

**Setup** To resemble the qualifying stage in the competition with 10000 gameticks, we chose to send 10000 packets during each test. The time waited between two packets was set to 1 ms, 10 ms and 20 ms. On the client side, we tested a simulated processing time of 0 ms (respond immediately), 2 ms and 4 ms. With these values, no timeout should occur. As a Windows test machine, we used a Core i7 930 running Windows 7 Enterprise 64bit. The Linux tests were performed on a compute server with an AMD Opteron 6276 (32 cores) running Ubuntu 12.04. Server and client were executed on the same machine using the loopback device.

**Results** Tables I and II show the results of the first test. Linux seems to properly handle the timeout of 10 ms: Only one occurrence was observed and this correctly happened after 10 ms. On Windows it is the complete opposite: Timeouts are generated way too early, sometimes after just 26 µs. On Windows it is the complete opposite: Timeouts are generated way too early, sometimes after just 26 µs.

<table>
<thead>
<tr>
<th>Wait time</th>
<th>Client 0 ms</th>
<th>Client 2 ms</th>
<th>Client 4 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 20 ms</td>
<td>1 (0.026 ms)</td>
<td>1 (1.72 ms)</td>
<td>1 (4.14 ms)</td>
</tr>
<tr>
<td>Server 10 ms</td>
<td>1 (0.064 ms)</td>
<td>1 (0.6 ms)</td>
<td>1 (0.42 ms)</td>
</tr>
<tr>
<td>Server 1 ms</td>
<td>1 (0.059 ms)</td>
<td>594 (all before 10 ms)</td>
<td>1605 (ditto)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Wait time</th>
<th>Client 0 ms</th>
<th>Client 2 ms</th>
<th>Client 4 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server 20 ms</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Server 10 ms</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Server 1 ms</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

While this behavior makes Windows look bad, it conforms to the documentation [4], which specifies the timeout as the maximum time to wait. By checking the counter of the last packet received, we found out that each timeout introduces a lag in the connection, independent of the operating system. In the most extreme case, the server processed packet number 8395 when waiting for the reply to sensordata number 10000. This implies that all packets which arrive too late are forever kept in the input buffer.

Summarized, the current implementation leads to the following undesirable effects:

- A controller is never able to recover from a timeout. This affects both Linux and Windows.
- On Windows, the implementation of the timeout does not work. Timeouts are often generated too early. The situation gets worse, when the calls to *select* happen very fast. This is the case in textmode, when the game does not spend time to render graphics.

One could argue that the first point is *ok*, because it “punishes” controllers which are too slow. But as we observed during our tests, timeouts also happen although the client reacts fast enough. And timeouts can also be caused by a lot of outside factors which cannot be influenced by a controller at all, e.g. machine load, scheduling behavior of the operating system, an unreliable network, etc. It therefor seems to be only fair to make sure that the server always tries to process the latest response of the client.

### III. Suggested Improvements

To solve the problems described before, a possible solution has to achieve two things: Make sure that the server definitely waits for the defined time (by default 10 ms) and make sure that the server can identify the right packet. We therefore propose

1) to constantly poll the socket instead of using the timeout parameter of the *select* function and

2) to extend the text protocol by a packet or gametick counter.

Polling could be implemented using the *select* function but since *select* is a bit clumsy to use (initialization of the *fd_set* structure), we suggest to use the *ioctl* (Linux) or *ioctlsocket* (Windows) function. Both functions, although having different names, behave exactly the same way and can be used to query the number of bytes stored in the input buffer of a socket. If the function reports that a non zero amount of data is available, it is guaranteed that the next call of the *recv* function will not block. By checking the counter contained in the received data, the server can decide if this packet is the one it is waiting for or if it is an older packet, which might be dropped if a newer packet arrives in time.

We tested this approach and it indeed solves all the problems. An implementation, along with the test server and client mentioned earlier, is available online [5]. This package also contains a modified version of the SCRC server bot.

### IV. Summary and Conclusions

We reported and analyzed some shortcomings of the current implementation of the SCRC software interface. These are solved with the suggested improvements which have already been integrated into the SCRC server bot. We hope that this modification can be incorporated into the official SCRC bot and that the SCRC will take place again in the near future.

### References


