UNIVERSITY OF OXFORD
SOFTWARE ENGINEERING PROGRAMME

Wolfson Building, Parks Road, Oxford OX1 3QD, UK
Tel +44(0)1865 283525 Fax +44(0)1865 283531
info@softeng.ox.ac.uk www.softeng.ox.ac.uk

Part-time postgraduate study in software engineering

Concurrent Programming, CPR

25th – 29th January 2021

ASSIGNMENT

The purpose of this assignment is to test the extent to which you have achieved the learning objectives of the course. As such, your answer must be substantially your own original work. Where material has been quoted, reproduced, or co-authored, you should take care to identify the extent of that material, and the source or co-author.

Your answers to the questions on this assignment should be submitted using the Software Engineering Programme website — `www.softeng.ox.ac.uk` — following the submission guidelines. When submitting the assignment online, it is important that you formally complete all three assignment submission steps: step 1, read through the declaration; step 2, upload your files; step 3, check your files.

The deadline for submission is 12 noon on Tuesday, 16th March 2021. You are strongly encouraged to submit a version well before the deadline. You may update your submission as often as you like before the deadline, but no submissions or changes will be accepted after the deadline.

We hope to have preliminary results and comments available during the week commencing Monday, 26th April 2021. The final results and comments will be available after the subsequent examiners' meeting.

ANY QUERIES OR REQUESTS FOR CLARIFICATION REGARDING THIS ASSIGNMENT OR PROBLEMS INSTALLING SOFTWARE SHOULD, IN THE FIRST INSTANCE, BE DIRECTED TO THE PROGRAMME OFFICE WITHIN THE NEXT TWO WEEKS.

Introduction

Through this assignment, you will demonstrate your understanding of concurrency, scalability and fault tolerance in Erlang-based soft real-time systems. You will do so by implementing an online auction SAAS offering and extending it with support for fault tolerance and distribution.

The assignment is divided into a practical and a theoretical section. The practical section is a coding exercise for a concurrent, soft real-time application. The theoretical section consists of questions on the application that you implemented, as well as other concurrency related items covered during the lectures.

Deliverables

While the practical deliverables are related, hand in a separate version of the source code for each exercise demonstrating the intermediate steps before achieving the final result. Do not submit your code in word or pdf format! Include an architectural description with the practical assignment, examples how to run it, outlining assumptions you have made. Use your report to show you understand the challenges and problems involved in concurrent programming. Make sure you are strict in respecting the provided APIs. The practical assignment accounts for the majority of the grade, where the quality of the code is as important as your report.

For Part 1 (Practicals), hand in a report in PDF format and a single .tar or .zip file containing the source code files corresponding to each deliverable which could consist of one or more modules. The deliverables are:

- The auction repository module
- The auction engine module
- The Pub/Sub module
- The distributed client module
- Any files containing your test data

For Part 2 (Theoretical), hand in a single PDF representing your writeup. Please ensure you stay within the word limits specified.

The Exercise

The need for scalable and reliable online auctions has increased as the result of COVID19 forcing existing in person auction companies to get creative. In turning around a negative, you become the founder of a

startup developing a SAAS offering auction houses can integrate in their existing websites. You will initially be implementing the backend system, allowing auction houses to create and run online auctions, allowing their users to browse and bid on items.

The server will consist of a service API, allowing the auction house to
- create
- populate
- start an auction

The client API, where allows users to

- browse items by auction,
- bid on them,
- subscribe to receive notifications from ongoing auctions.

We will break the exercise up into smaller parts that are easier to develop and test, combining them in a supervision tree.

Practical Part 1: An Auction Repository

The first part of the system will allow you to create auctions, add/remove items to an auction and get the items and their starting bids for an auction.

Implement following functions in the **`auction_data.erl`** module, it should create an auction and all associated data structures for it:

```
auction_data:create_auction() -> {ok, AuctionId}.
```

Create a function to add items to a given auction. Give some through on what datatype to use for your ItemId:

```
auction_data:add_items(AuctionId,[{Item,Desc,Bid}]) ->
  {ok, [{ItemId, Item}]} | {error, unknown_auction}.
```

You also will need functions that retrieves the list of items for an auction and another one to see the details for a particular:

```
auction_data:get_auctions() -> {ok, [AuctionId]}.

auction_data:get_items(AuctionId) ->
  {ok, [ItemId]} | {error, unknown_auction}.
```

The IdemId list is returned in lexicographical order.

```
auction_data:get_item(AuctionId, ItemId) ->
  {ok, {ItemId,Desc,Bid}} |
  {error, unknown_item | unknown_auction}.
```

Don't forget that we need functions to remove items:

```
auction_data:remove_auction(AuctionId) ->
  ok | {error, unknown_auction}.

auction_data:remove_item(AuctionId,ItemId) ->
  ok | {error, unknown_item | unknown_auction}.
```

Hint: You can use `erlang:make_ref/0` to generate your AuctionId. You should also consider carefully what type ItemIds should be for this API, and make sure they work in a distributed environment.

Practical Part 2: The Auction Process

The auction itself is handled by a server implemented in the **`auction.erl`** module. Add a function to start an auction that uses the ID of an existing auction:

```
auction:start_link(AuctionId) ->
  {ok, Pid} | {error, unknown_auction}.
```

Once an auction starts, the auction data cannot be modified, so make sure you lock it in your repository code. The process will start auctioning in the same order the function `auction_data:get_items/1` returned them in, ordered in lexicographical order based on the ItemId.

Users will be able to bid on the current item and, to avoid mistakes, will have to specify which item they intend to bid on, not just the `AuctionId`. Add the function that handles the bids and tells the user if they are leading the auction or not:

```
auction:bid(AuctionId, ItemId, Bid, Bidder) ->
  {ok, leading | {not_leading, CurrentBid}} |
  {error, invalid_auction |
          invalid_item |
          auction_ended |
          item_already_sold}.
```

Wait for ten seconds of inactivity for an opening bid or a new leading bid before moving on to the next item. Be careful, this can be tricky.

Hint: `item_already_sold` and `auction_ended` means the system needs to store information about past auctions even after they finished, and who won the bid.

Practical Part 3: Pub Sub Engine

Implement a publish - subscribe engine which allows clients to subscribe to auction specific events. Put your code in a library module called **`pubsub.erl`.**

Start the pubsub engine linking to its parent:

```
pubsub:start_link() -> {ok, Pid}
```

Create and delete channels, where every `Channel` is an auction. Channels are used to broadcast events relating to a particular auction:

```
pubsub:create_channel(Channel) ->
  ok | {error, duplicate_channel}

pubsub:delete_channel(Channel) ->
  ok | {error, unknown_channel}
```

Users can subscribe to auctions only if the channel has been created. When a channel is deleted, all users are unsubscribed.

```
pubsub:subscribe(Channel) ->
  ok | {error, unknown_channel}

pubsub:unsubscribe(Channel) ->
  ok | {error, unknown_channel}
```

At this point, implement the function which sends the notifications to the subscriber. Use the `Pid` of the process that called `pubsub:subscribe/1`, sending it `Event` as a message:

```
pubsub:publish(Channel, Event) ->
  ok | {error, unknown_channel}
```

Now, connect the auction module to your publish - subscribe engine,

creating channels when auctions are created and generate events:

```
auction:subscribe(AuctionId) ->
  {ok, MonitorRef} | {error, unknown_auction}.
```

After that function is called, the subscribed process should expect the following update messages from the pubsub process:

```
{auction_event, auction_started}
{auction_event, {new_item, ItemId, Description, Bid}}
{auction_event, {new_bid, ItemId, Bid}}
{auction_event, {item_sold, ItemId, WinningBid}}
{auction_event, auction_closed}
```

Practical Part 4: Implementing The Client

You can now implement the client which interfaces with the backend server, receiving and reacting on notifications. As different auction houses can integrate to the API as they please, we will leave it up to you to decide how to implement it. It should be able to run your client code on a node different from the back-end server using distributed Erlang, and provide feedback on ongoing bids using `io:format/2`.

Be creative about the functionality you can provide your users, such as automated increase in bids automatically increasing it by a certain percentage or amount, handling race conditions where other users place a higher bid before yours is received, or implementing agents which decide on bid increases based on the behaviour of the other bidders. Your imagination (and time available) is the limit.

Document how your client works and how it can be run in your report.

Theoretical Part:

Answer each question with a maximum of 200 words. If needed, feel free to use diagrams to support your answers. Short concise answers are preferred to long ones.

1. We have not implemented any fault tolerance in your system. How would you model dependencies in a supervision tree and make sure there is no single point of failure in your system?

2. There are race conditions in your system, such as (but not limited to) identical bids arriving at the same time. How are they handled?

Can they be exploited?

3. How does a shared memory concurrency model (such as threads) compare and contrast to a no shared memory model (such as the actor model or Erlang style concurrency)?

4. How do you handle bottlenecks when working with no shared memory concurrency models?

Guidance

In the practical section, your solution should only use libraries from the Erlang/OTP distribution. Make sure you follow the interface descriptions! Demonstrate your code with the given test cases. The test cases act as a basic benchmark for requirement satisfaction, but aren't representative of all the things that can happen or go wrong in a live production system, especially borderline cases.

You may be able to find partial solutions to the problems on the web. I recommend that you don't use or rely on them: they are often of dubious quality, they are likely to implement slightly different specifications, they typically won't help your critical review, and they won't help you learn about concurrent programming. Whatever you do, do make sure you make clear the source and the extent of any derivative material and demonstrate a clear understanding of the problem in your report.

Finally, please structure your answer so that there is a cover sheet that contains *only* your name, the subject and date, and a note of the total number of pages. Do not put any answer material on the cover sheet; begin your answer on a fresh page. Avoid putting your name on any page except the cover page. Please number the pages and sections. Include a soft version of your code with your assignment report.

Assessment criteria

The practical part of the assignment consists of 70% of the final grade and is intended to evaluate:

- Your ability to implement highly concurrent and distributed real-time systems in Erlang;

- Your ability to express algorithms using appropriate data structures and elegant function definitions;

- Your ability to write clear and simple code that is both scalable and fault tolerant.

The assessment of the theoretical question will count towards 30% of the final grade. You will be graded based on the clarity and conciseness of your answers. If you do not keep within the word limit, points will be deducted.