SOFTWARE ENGINEERING PROGRAMME
UNIVERSITY OF OXFORD
www.softeng.ox.ac.uk

ASSESSMENT

Student: Nicholas Drake

Course: Concurrent Programming

Date: 25th January 2021

Grade: 85

Francesco Cesarini

June 10, 2021

REPORT

A solid practical and theoretical solution which could be used as a model answer The application is complete, as is the quality and extent of the explanation of your approach. The report gives a very detailed description of the system. For each section it describes various alternatives for how they could be implemented and discusses the pros and cons of each and why the ones were chosen. It then describes the implementation of each. Unfortunately there is no direct overview of how the parts interact with each other, though supervision trees for the system are shown. The code is well documented and OTP is used in all the sections. There is an extensive set of test cases built on Common Test, which shows a good initiative at wanting to go beyond what is required. There are types and functions specs for all the code in the system. Well done, as we did not cover these parts in class. The system has been designed to be run with rebar3 and has been structured accordingly. It also uses the supervision trees described in the documentation.

3.1 Practical Part 1: An Auction Repository

In this section, we are looking for a process which uses some form of storage to save auction data. Calls to write the data can be either synchronous or asynchronous, whilst retrieving data have to be synchronous. We would expect the process to link to its parent process. You are using mnesia to store the auction data. It is a practical way of ensuring that the data is consistently saved on disc on all the nodes taking part in the system.

3.2 Practical Part 2: The Auction Process

In this section, we are expecting a process implementing either a server or a finite state machine. We look at how you address timeouts after an auction, and if the approach can be abused (e.g. is the timer external and triggers a message after ten seconds or does flooding the system with invalid requests reset it?). Using gen_statem for auction processing is interesting and an easy way to keep track of when the auction has ended.

3.3 Practical Part 3: Pub Sub Engine

In this section, we are testing your ability to think concurrently. What happens when the same module runs in multiple processes, how do you name the process or manage restart? Or is your implementation a single process

handling multiple channels, and if so, does it risk becoming a bottleneck? In your solution, you use event managers/handlers to implement the pub-sub service. It is an interesting idea and gives another way of managing the channels and their subscribers.

3.4 Practical Part 4: Implementing the Client

Here, we were testing your ability to follow the APIs provided by the other processes, and test your understanding of client-server principles. By being intentionally vague, we were looking for creative solutions which would highlight the best features of the system you implemented. You have implemented a good way of handling the clients using a supervisor.

Theoretical

4.1 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?
When implementing supervision trees, you need to think of the order in which your processes are started, and how they are dependent on each other, and how you group processes together in case of failure. A well structured supervision tree for the auction SAAS offering would include on the top level (started in this order):
A behaviour for the auction data

* A behaviour for the pubsub server
* A supervisor handling behaviours, each implementing a single auction
* Depending on the implementation of your pub-sub server, you could have a pubsub supervisor and represent each channel as a process.

If an auction process terminates, you restart it. If a pubsub server terminates, you restart the server and the auction supervisor (which restarts the auctions). If the auction data server terminates, you restart everything, as both the pubsub and the auction processes are dependent on it. In OTP speak, this is a rest-for one strategy. In your answer, if a pub-sub process terminates and is restarted, you might miss some of the auction events. Would it not be better to also restart the auction process?
No single point of failure means two of everything. This puts a requirement for you to replicate the state on another node, so if a node terminates, it will be possible to recreate the same supervision tree on a standby node. How this is done is up to you, it could be with Mnesia or just a simple replication

layer you implement yourself. The supervision tree on its own, as you state, is a single point of failure.

4.2 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?
Well answered, exactly what I was looking for. Bids arriving at the same time should not be a problem, as they are serialised in the auction process mailbox and handled sequentially in the order in which they are received. It makes it hard to exploit, even when flooding the system with invalid requests, as the valid ones are handled in the order in which they are received. Timeouts could be triggered externally, through a separate process, and a message created after a configurable timeout for the auction. This means that all invalid messages, and possibly valid ones with higher bids sent within the timeout time are handled. Valid messages would reset the timeout, addressing any exploitation attempts.

4.3 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)?
Well answered, you covered almost everything I was looking for: Systems based on shared memory concurrency models run on a single machine, and are often extremely fast, but limited in how much they can scale. They are ideal for parallelism and computationally intensive activities. No shared memory approach to concurrency results in a model where processes communicate with message passing. Location where these processes are located becomes irrelevant (whilst latency is affected), allowing them to be placed on separate cores and/or CPUs. So whilst shared memory models are built for speed, no shared memory models are built for scale and fault tolerance.

4.4. How do you handle bottlenecks when working with no shared memory concurrency models?
Bottlenecks manifest themselves in many different ways, and as such, can be handled differently. Using the memory/0 BIF might help, but there might be easier ways. To address bottlenecks. you should try to reduce the amount of work which is done in the server process where there are bottlenecks, moving as much as possible to the client process. If producers create requests at a rate faster than what consumers are able to handle, a possibility is to throttle producers using a variety of techniques, including synchronous requests, where a new request is not sent until the previous one has been handled. If bottlenecks persist, load regulation and back pressure needs to be put in

place ensuring that only a sustainable load is applied to the system.