

# Concurrent Programming

Stefan Cross

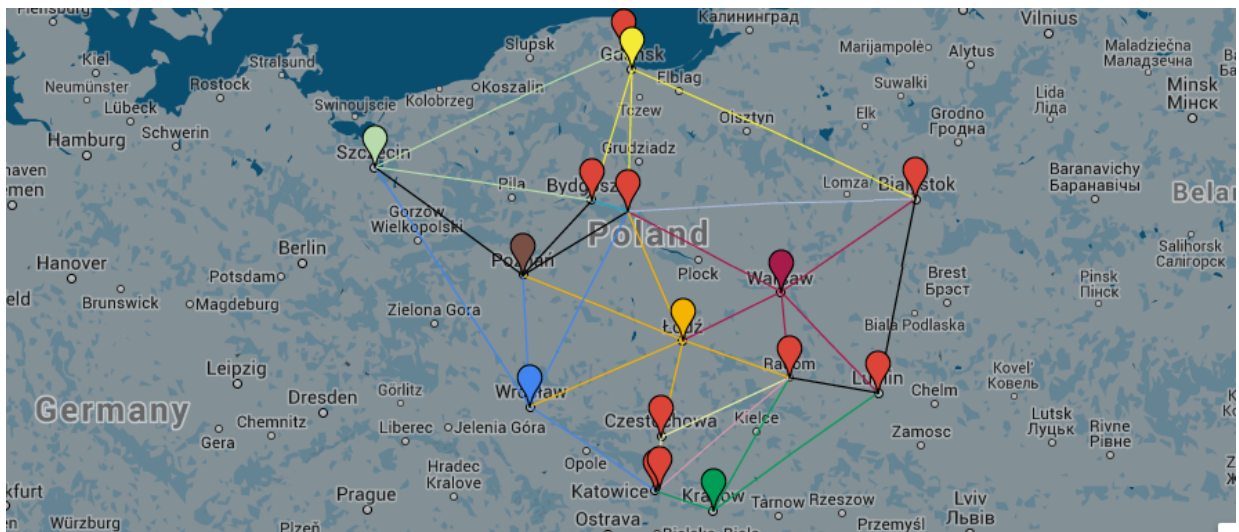
*Michaelmas Term, 2014*

# Introduction

This paper deals with a Courier Delivery Problem (CDP) in a Discrete Event Simulation (DES) addressing concurrency, fault tolerance and scalability issues inherent in the problem in Erlangs soft real-time systems. The exercise implements a nationwide parcel delivery simulator used to optimise delivery times and reduce transport costs. The simulator runs off a configuration file that specifies a list of locations and populations, distances between these locations, and the starting locations of a fleet of vehicles which have different capacities.

The simulator will consist of a dispatcher informing the vans and trucks where to pick up and deliver parcels, as well as a journey planner, telling them which routes to take in-between cities. When the simulator is up and running, the system is ready to take in new parcel delivery orders. By testing different dispatch strategies, routes between cities and vehicle behaviour will provide metrics which allows us to optimise the service in relation to delivery times, capacity and fuel consumption.<sup>1</sup>

All code and working is provided in the same folder that this document was delivered in and will be available online with version control and supporting instructions in a README.md file. This file will also include examples of test cases and also an explanation of each sections implementation and design decisions.<sup>2</sup>



The map above illustrate visually the specifications configuration locations and mappings that may be useful to refer back to later in the paper.<sup>3</sup>

<sup>1</sup> Taken from the specification document CPR\_2014-09-27-OxfordAssignment+coversheet.pdf

<sup>2</sup> <https://github.com/stefan-cross/CPR2014>

<sup>3</sup> [https://www.google.com/maps/d/edit?mid=zZqQW5ER\\_nEU.k0nVn\\_MpDiol](https://www.google.com/maps/d/edit?mid=zZqQW5ER_nEU.k0nVn_MpDiol)

## *Practical Part 1: Implementing a Journey Planner*

Please refer to the source code folder “Part 1” for the implementation and also the README.md file for instructions on execution.

The journey planner was implemented as a generic server but makes use of the Digraph library to handle the logic of routing finding between locations. It was decided to return nested loop of “From”, “To” so that distance might be able to be included at a later date and allow for optimised routing on link cost rather than hop count, however there was not enough time for this. Although this routing technique is not efficient it is reliable and there were other areas more integral to the CDP to address.

The input to `planner:route/2` uses `lists:usort` removes duplicates effectively, however this may not lead to the most efficient route. Potentially this could be remedied by removing such duplicates but some test cases are particularly unfortunate due to the `lists:usort` and that these locations are not alphabetically contiguous and I have opted simplicity over complexity with random efficiency. However the aim of this exercise is to flex the power of Erlangs actor concurrency model rather than weigh up hop and distance vector based routing algorithms. We can make gains by can simply test package eligibility to be dropped when in each city and calling `manager:reserve` whenever the vehicle is empty.

After running simulations with random weighted packages it also appears that the capacity of vans is such that they can not take significant load and likely to only have one destination, and equally larger loads in the trucks are often more efficient when we select deliveries by location and time so that we have a higher quantity to drop in just one location, thus rendering an optimised routing function less necessary. The loop functionality was added later to permit for message passing to ascertain a route.

## *Practical Part 2: Implementing the Parcel Manager*

Please refer to the source code folder “Part 2” for the implementation and also the README.md file for instructions on execution.

The manager module keeps track of parcels and provides vans and trucks with locations that require pickups. It can access the journey planner's tables and retrieve information on cities and routes as required. Note that the setup items have been refactored out of the planner and also combined them with the additional supporting set up operations in its own setup file where ETS tables are created, the config is imported, the graphs created and the planner started in support preparation.

Sending items allows a client to specify a starting point and destination as well as the weight of the item and is returned a unique reference. This unique reference makes use of Erlangs `now()` built in function as this has the desirable feature of being a monotonic function that always increases and thus requires a global lock to read. This will prevent multiple identical references. This was tested with multiple concurrent orders being placed to ensure that no two references were the same.

The deliver function returns a list of near by locations from the current location of a vehicle with the oldest parcel being listed first to ensure the older the item the higher the priority. The reverse ETS select facilitates this functionality as opposed to a standard select which returns the newest order reference first. The specification requests this function returns a location list of destinations that have orders to be delivered to or picked up from, although it is not clear why you would want the locations to be delivered to without the concept of ownership of an order at this point.

The two reserve functions are similar and differ in the arity where one accepts the addition of a “To” destination location. Older orders are prioritised with the reverse select and it is ensured that only one vehicle can pick it up by updating the entry and stating the identification of the vehicle to which it is reserved.

The pickup function was relatively straight forward to implement however the introduction of the return value of instance was somewhat ambiguous despite some clarification defining this as “The parcel you are trying to do something with does not exist. For example, trying to drop a parcel you have not picked up or which as a result of a race condition, has been reserved and picked up by someone else.” It was therefore decided to return the failed reference identified as this instance.

The drop function is called when a vehicle is at its destination and follows the logical progression of pick/1 and given a reference will allow the update of the record to show that it has arrived at its destination.

The transit function permits a vehicle to drop an order at a cargo station en route to its final destination. This updates the corresponding order by reference in the manager ETS table and updates the "From" location to be the current cargo station location.

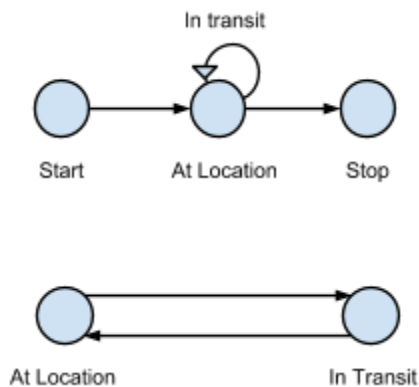
The cargo function is called when deliveries are complete and provided with a location the function will return the nearest cargo station, by location hop count, not distance based.

Finally the lookup/1 function allows you to query the state of a package in the system by the provided reference.

## Practical Part 3: Implementing the Vehicles

Please refer to the source code folder “Part 3” for the implementation and also the README.md file for instructions on execution.

The vehicle implementation is based around a finite state machine (FSM) design pattern. At this stage we do not concern ourselves with starting all the vehicles as specified in the config, instead creating a few individual cases and implementing the algorithm as per the specification.



A vehicle can simply be in one of two state, at a location, or in transit. There is also the processes start and stop state but in terms of the FSM we ensure that a vehicle is only in one of these two states.

It is worth noting that if a vehicle is started in a location that is not valid then it will still be created but never have the opportunity to change its state or location to participate in the simulation. Also if identical vehicle names are registered an error will be returned which is left just as “bad arguments” due to time constraints.

It was also found that various formatting functions had to be created to handle the output of the manager and planner functions which was not ideal, where possible the API for the previous modules has been respected but there are some slight variations to avoid having to pattern match on some outputs where a simple empty list would suffice.

The specification emphasised the priority of older orders and every effort was made to do so however this was difficult to track under high throughput as deliveries are initially selected by location. Another optimisation that I ran out of time to implement would have been to maximise vehicle capacity by giving the lower capacity vehicles priority on smaller packages, however this would conflict with the specification point about order timing priority. However this would be worth exploring as during tests it appeared that this was a bottleneck with inefficient vehicle loads.

At this point the ETS table structure has been altered to allow for a process to be noted next to an order, as simply updating a record to show it is reserved would not distinguish it from other process reservations. In order to still comply with the original specification API the manager functions were left relatively intact and instead you will find internal formatting functions to add in the additional process identifier to the order records in the manager ETS table. Although not optimal by any stretch it was felt that specification requested strict adherence to the API and so this was followed at detriment to performance.

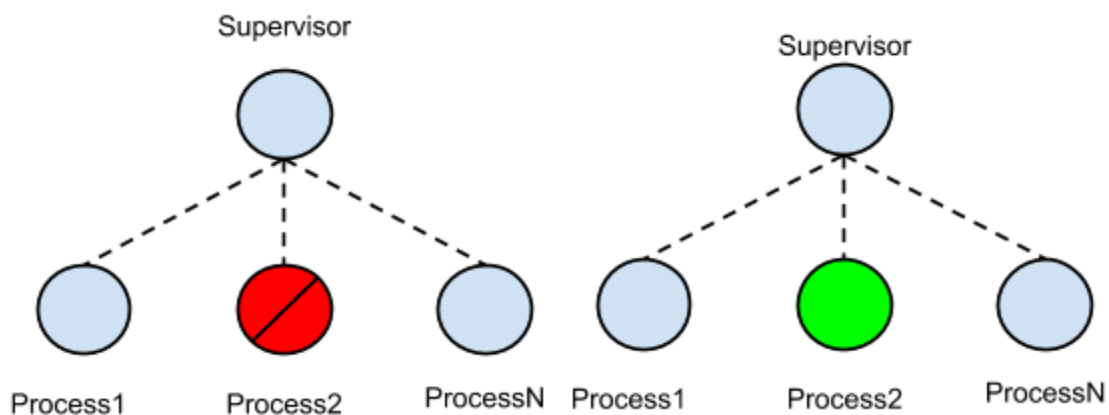
## Practical Part 4: Implementing Fault Tolerance

Please refer to the source code folder “Part 4” for the implementation and also the README.md file for instructions on execution.

This final part of the implementation is in two parts, first a vehicle supervisor which will handle the potential issues of vehicle nodes and the second part deals with higher level issues, dealing with the manager, planner and orchestration.

### 4.1 Vehicle supervisor

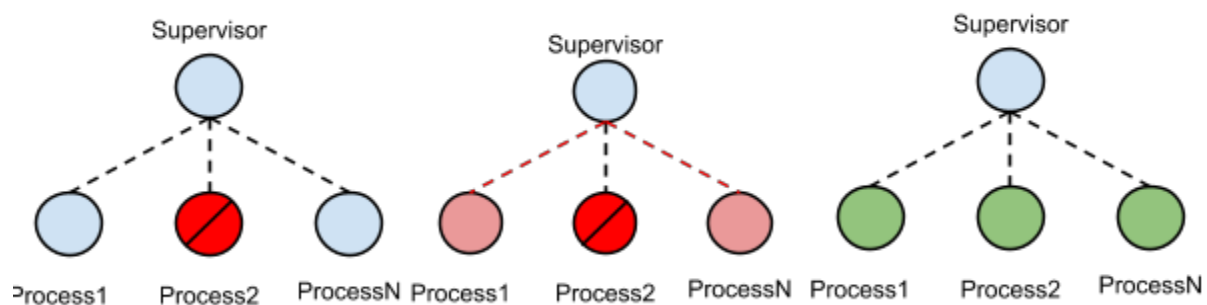
The vehicle supervisor traps errors from the child vehicle processes and return their contents to the main manger ETS table, eligible for circulation again, and then restart the vehicle in the last confirmed location. This supervisor process operates a one-for-one strategy, with just the



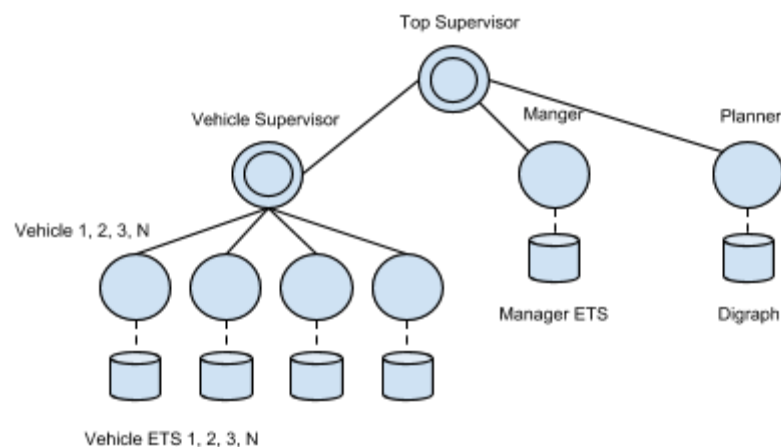
one supervisor to all the vehicle processes as illustrated below. Testing proves that the correct number of parcels remain in circulation irrelevant of how many time one might try and sabotage vehicles. Alterations to note here is that we have split out the ETS tables across the individual vehicle processes to try and reduce the load on just one ETS table. An important change to accommodate this was to make the vehicle table heirs of the manager, that way, in the event of an error the table is not destroyed. Disk Erlang Term Storage (DETS), a more persistent version of ETS was considered, however I opted for this change of heir to circumvent the issue of data loss on a process termination. We have also fully automated the creation of all the vehicles from the configuration file. To add more, we can simply add them to the file.config.csv.

## 4.2 Top level supervisor

This final part is the implementation of a top level supervisor. Open Telecommunication Protocol (OTP) modules are utilised to good effect. (OTP is considered to be three main components, a set of modules, design patterns and principles and Erlang itself.) In the event that one of the major components such as the planner, manager of vehicle supervisor failing, then this supervisor can act accordingly. It would have been ideal to strategise the significance of each component and deal with different combination in different means, alas time escaped me and the relatively primitive strategy was implemented.



In the event that the vehicles supervisor goes down, then they are all restarted again in a one for all fashion, should the planner go down then this is also started again in a one to one fashion. In the event that the manager terminates then we have lost all the orders, and therefore we will have to start all over again anyway. A rest for one strategy might suffice when the manager crashes and needs to take down the vehicles in order to restart effectively however the planner could remain intact. The final abstracted system design is illustrated below.





## Conclusions

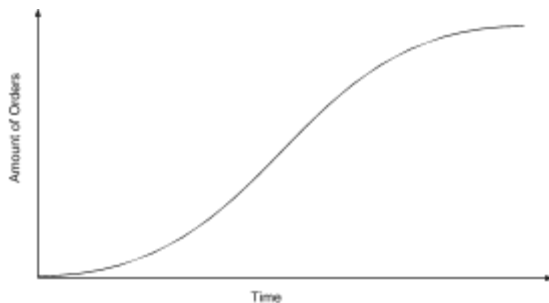
The main issue encountered when implementing this section was to adhere to the several specification points outlining the algorithm for vehicle interaction with in the system. In order to confirm to points there was significant overhead computing the value of dropping items in depots en route, the issue here was that the route was subject to change and it became increasingly difficult to debug. For this reason I adopt a more “brute force” approach in the where by when a vehicle is out of work then they are posted to a random location, en route they pick things up and then continue with the work. I found this approach easier to debug and the concept of depots brings little advantage when the routing algorithm is just hop count and not optimised to account for distance in the first place. Instead I focused on utilising Erlangs concurrent and distribution benefits which I felt more important than concentrating on the routing algorithms. The important features are still implemented in that only one process/vehicle can possess and order at any given time and all orders are eventually distributed to their destinations.

There are three distribution policies that were implemented. The first was to choose a random city and en route to said location look for work to pick up and adjust the route on receipt of a package. The second is to go to a cargo center looking for work en route, the idea being that they are more likely to contain packages if we had implemented the cargo priority drop as per the suggested specification. The third would be to prioritise the oldest orders first and not to accept any other work en route.

Unfortunately there is a heavy dependency on ETS and this would likely be the main bottleneck in the program. This was done in the effort to make the application more dynamic, and so that the simulation can be run from just the config data. ETS also allows for simple visualization of the data processing taking place, with some overhead, but otherwise useful. Below are some examples of using Erlang observer:start/0 tool to achieve some insight into the programs data processing power.



It was observed that the amount of orders in the system had an impact on the speed of distribution. With only 10,000 orders and 12 vehicles processing the throughput was ~100 per second, whereas when there were 100,000 orders and 12 vehicles there was a dramatic drop off to ~10 orders processed a second. Increasing the orders to 500,000 and even 1,000,000 emphasised this further.



When processing 100,000 orders the overhead of the storage is not too detrimental, and performance is still acceptable up to 500,000 orders, however there is significant drop off at around 1,000,000 orders as the tables become unwieldy.

During the implementation tried to adhere to best practices, avoiding sequential logic and breaking this out into separate function to utilise pattern matching, reducing interdependencies, and hiding message passing in functional interfaces where possible, adding documentation and even some unit testing. There is evidence of higher order constructs, meta programming with config import and functions as arguments, “fun” for anonymous spawn functions and functions as results. Where possible I have tried to conform to stylist conventions to increase readability; avoiding nesting, and reducing side effects to the same module. Some more verbose assignments were made to ease the eligibility of code.

Given more time i would have investigated better utilisation of the concepts of depots and a weighted routing function, not just hop count. It would have also been interesting to visualisation on map. Further investigation into the use of Menisa and full distribution across multiple nodes using multiple managers for full distribution would be interesting as well as applying more rules from the configuration file in conjunction with meta programming to change behaviour of program depending on the config. This might permit a self correcting and optimising program to work out the optimum placement of cargo stations and more realistic simulations with more orders originating from more populated locations or where the location has significant relevance such as Gdańsk as a shipping port that would likely affect a real system and order distribution.

It would have been desirable to refactor the code towards the end to utilise selective receive and to do away with some of the obtruse formatting functions evident in the vehicle module. It would have also been a cleaner implementation if i had leverage some more OTP patterns which I only became comfortable with when implementing the top level supervisor, however it was beyond the scope of this paper.

## 1. When dealing with concurrency in distributed systems, what are the advantages of having asynchronous message passing?

Asynchronous message passing is fundamental to the Actor concurrency model<sup>4</sup>, appropriate for concurrent tasks requiring high responsiveness in problem domains with large amounts of communication between processes. This non blocking state circumvents the need for shared memory and increases reliability. Non blocking processes do not wait for the acknowledgement of a message, they can continue operations on safe remote function calls because there are no assumptions about the consequences,<sup>5</sup> and thus can increase throughput. It also improves fault tolerance in error handling and facilitates vertical and horizontal scalability making better use system architecture and also distributing across nodes effectively.

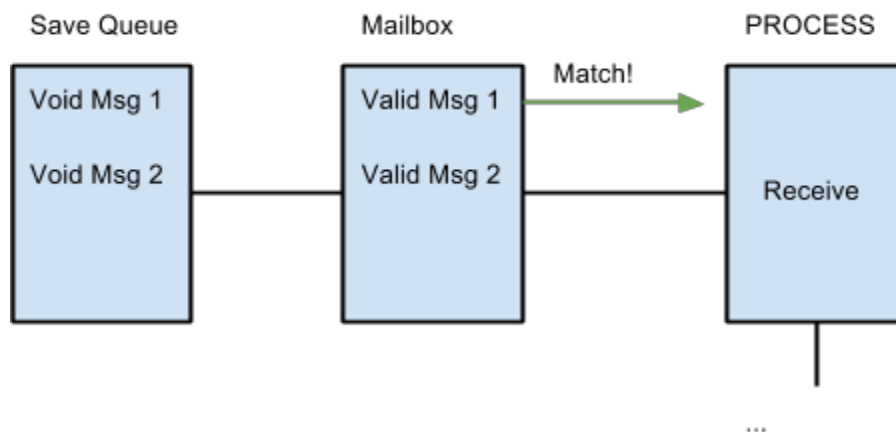
---

<sup>4</sup> [http://en.wikipedia.org/wiki/Actor\\_model#Unbounded\\_nondeterminism\\_controversy](http://en.wikipedia.org/wiki/Actor_model#Unbounded_nondeterminism_controversy)

<sup>5</sup> <http://learnyoussomeerlang.com/the-hitchhikers-guide-to-concurrency>

## 2. What is the advantage of a selective receive in your concurrency model when implementing finite state machines?

It facilitates efficient pattern matching by listen to for specific formats and discard the noncompliant messages without need to explicit guards. Rather than writing a multitude of defensive style guards it was possible to only receive the required format. It enables us to rise above the defensive programming mantra and discarding the irrelevant and concentrating the concerning messages passed to a processes which reduces the codomain of a function. This leaves our state machine logic much cleaner and concise as we only have to handle specific messages that will alter state.



### 3. How would you stress test your system in order to find bottlenecks? How would you detect these bottlenecks?

Using various amounts of orders and vehicle worker processes I observed the strain on resources by monitoring Erlangs observer tool. The simulation relies heavily on the use of ETS. As the system evolved it became apparent that one central table holding all the orders and updating records as they passed through various states was stifling. Instead it was opted to keep the initial orders in a tab and have individual tabs of the vehicle contents as well as a delivered tab that would transition packages through their journey and distribute a single ETS among N number of processes acting as vehicles.

#### 4. Describe concurrency-related race conditions that can occur in your system, explaining how you have (or have not) addressed them.

When investigating concurrency settings on the ETS tables I altered the table structure to be a duplicate bag I noticed orders were being replicated. The volume of requests coming in to the ETS table and across a dozen concurrent processes would often attempt to process the same order. As the ETS calls are primitive and separated in to reads and then writes it permitted a number of pids to access the record before the update and thus allow multiple vehicles to simulate delivery of the same package. This was partially addressed by using ets:update over separate ets:read and ets:write.

Message passing utilises process mailboxes, serialising requests and thus only allowing one process access at a time to a record. However due to time constraints this was not implemented and instead the tables were altered to be sets and not bags, but this is only hiding the issue and not truly addressing the underly deadlock issue.

Other race conditions appeared when starting up the system. The following occurred when setting up the vehicle processes either in the register spawn method or loop. This was resolved to separating it out to a orchestration module.