Concurrency in Go and Erlang

Frank Braun

2020-08-07

A comparison between the concurrency mechanisms in Go and Erlang.

Lightweight processes

Both Go and Erlang have concurrency mechanisms inspired by communicating sequential processes[1]. They both have very lightweight concurrent *processes* which are not necessarily running in parallel. That is, they can run concurrently on a single core without the need to manually yield to another process. The scheduler of the language runtime does take care of that. If more than one core is available, processes in both languages run in parallel.

Starting a lightweight process in Go (called *goroutine*) for a function `f`:

```
go f()
```

In Erlang `spawn`[2] is called with the function to execute as the new process and it's argument:

```
Pid = spawn(Module, Function, Args)
```

It returns a *process identifier*.

So processes are actually quite similar in Go and Erlang, just the syntax is different.

Message Passing

Message passing between processes is different in Go and Erlang. In Go there is the explicit concept of *channels* (a type) that can be used to communicate between concurrently executing functions. Channels can be *unbuffered* or *buffered*. Sending to an unbuffered channel or a full buffered channel blocks the sender. Receiving from an unbuffered channel without a sender or from an empty buffered one blocks as well. Reading from channels works in a strict first in, first out

[1] https://en.wikipedia.org/wiki/Communicating_sequential_processes
[2] https://erlang.org/doc/man/erlang.html#spawn-3

manner. Since channels in Go are "first-class citizens" they can be assigned to variables, passed around in channels, etc.

In Erlang every process has an implicit *message queue* which can be send to by using corresponding the process identifier:

```
Pid ! Msg
```

Sending to a message queue never blocks the sender. Message queues are buffered with an "unlimited buffer" (until the memory limit is reached).

Reading from the buffer works by pattern matching of different clauses against the messages. If an earlier message doesn't match any clause it will be left in the queue and a later (matching) message is processed instead. That is, messages are not processed strictly in a first in, first out manner.

Message queues are not "first-class citizens" in Erlang.

Error handling

Go and Erlang differ massively in error handling.

In Go errors are supposed to be handled *explicitly*, exceptions (created by calling `panic()`) should be avoided. Exceptions can be handled[3] by `recover()`, but this should be avoided as a general error handling mechanism. That is, in general errors between different processes are explicitly passed around through channels.

Erlang, on the other hand, has a "crash early, crash often" philosophy. Corner cases (for example in pattern matching) are not explicitly handled, leading to a crash of the corresponding process. Processes are organized in hierarchies (for example, by using the generic supervisor[4] behavior) where supervising processes react to messages generated by crashing child processes.

Distribution

Go has no explicit support to distribute processes over multiple nodes. Although it has necessary networking capabilities in it's standard library, the process and channel primitives do not work across nodes.

Erlang has the capabilities to distribute[5] a software system over multiple nodes. That is, processes can be started on remote nodes and can be communicated with as with processes on local nodes. However, the distribution over multiple nodes is not something that happens automatically, it has to be built into the software explicitly. Furthermore, Erlang assumes a friendly network: Node communication is authenticated (by a shared secret), but not encrypted. So for

[3] https://golang.org/ref/spec#Handling_panics
[4] https://erlang.org/doc/man/supervisor.html
[5] https://erlang.org/doc/reference_manual/distributed.html

a node cluster on the Internet the network layer encryption would either have to be added in Erlang itself or the nodes have to be connected in such a way that the network layer is encrypted already.