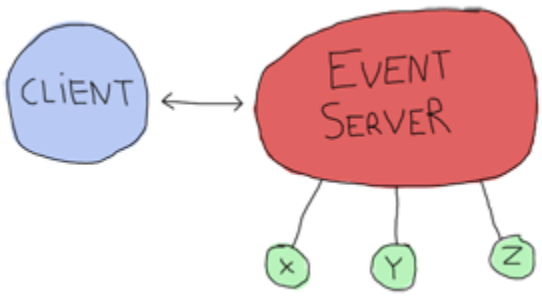
UNDERSTANDING THE PROBLEM

The first step is to know what the hell we're doing. "A reminder app," you say. "Of course," I say. But there's more. How do we plan on interacting with the software? What do we want it to do for us? How do we represent the program with processes? How do we know what messages to send?

As the quote goes, *"Walking on water and developing software from a specification are easy if both are frozen."* So let's get a spec and stick to it. Our little piece of software will allow us to do the following:

- Add an event. Events contain a deadline (the time to warn at), an event name and a description.
- Show a warning when the time has come for it.
- Cancel an event by name.
- No persistent disk storage. It's not needed to show the architectural concepts we'll see. It will suck for a real app, but I'll instead just show where it could be inserted if you wanted to do it and also point to a few helpful functions.
- Given we have no persistent storage, we have to be able to update the code while it is running.
- The interaction with the software will be done via the command line, but it should be possible to later extend it so other means could be used (say a GUI, web page access, instant messaging software, email, etc.)

Here's the structure of the program I picked to do it:



Where the client, event server and x, y and z are all processes. Here's what each of them can do:

Event Server

- Accepts subscriptions from clients
- Forwards notifications from event processes to each of the subscribers
- Accepts messages to add events (and start the x, y, z processes needed)
- Can accept messages to cancel an event and subsequently kill the event processes
- Can be terminated by a client
- Can have its code reloaded via the shell.

client

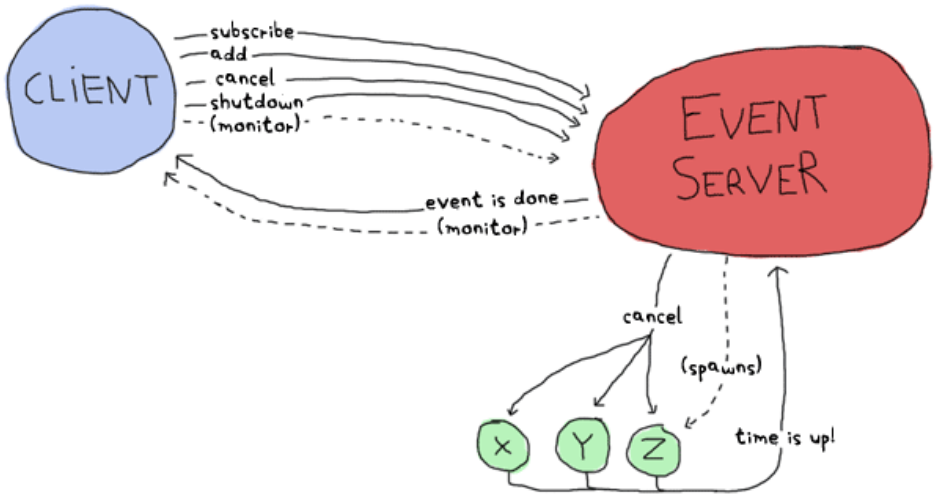
- Subscribes to the event server and receive notifications as messages. As such it should be easy to design a bunch of clients all subscribing to the event server. Each of these could potentially be a gateway to the different interaction points mentioned above (GUI, web page, instant messaging software, email, etc.)
- Asks the server to add an event with all its details
- Asks the server to cancel an event
- Monitors the server (to know if it goes down)
- Shuts down the event server if needed

x, y and z:

- Represent a notification waiting to fire (they're basically just timers linked to the event server)
- Send a message to the event server when the time is up
- Receive a cancellation message and die

Note that all clients (IM, mail, etc. which are not implemented in this book) are notified about all events, and a cancellation is not something to warn the clients about. Here the software is written for you and me, and it's assumed only one user will run it.

Here's a more complex graph with all the possible messages:



This represents every process we'll have. By drawing all the arrows there and saying they're messages, we've written a high level protocol, or at least its skeleton.

It should be noted that using one process per event to be reminded of is likely going to be overkill and hard to scale in a real world application. However, for an application you are going to be the sole user of, this is good enough. A different approach could be using functions such as timer:send_after/2-3 to avoid spawning too many processes.

Source : http://learnyousomeerlang.com/designing-a-concurrent-application