Comparative Programming Languages
Exercise Session for Erlang

October 24, 2014

Important

Please, inspect the `homework` document for general remarks, compilation, execution and terminology related to Erlang and these exercises.

1 Exercises

This exercise session will have you expanding the process created for homework: *the killer*. Should you not have done the homework, start by doing that first.

In this exercise session you will create more complex processes that interact with some of the code provided on Toledo. In this batch you will need to run a different server for each exercise in order to test your code—you will be writing the client. Start the server by typing the corresponding call in the console window or on the command line in an `erl` shell. At the beginning of each exercise you will find which server you have to start and the name (atom) it is registered as. This atom will be used to send messages to the server.

1.1 Failing messages

You should have a simple attacking process (the *killer*). Make a copy of it and prepare to expand its functionality.

In this exercise the killer must interact with an intermittent server, e.g. a server that sometimes is reachable and sometimes not. Execute `srv:startI()` to run the server. The server will be registered (and unregistered) locally via the name `intermittent`. The server will be alternating between the two states, sometimes sending messages to the server will fail.

Expand the killer so that it is able to send messages to the intermittent server without crashing. Notice that if you send a message to a name that is invalid (an unregistered atom or an non-existent PID), the message send will raise an exception. Dealing with exceptions in Erlang is like in Java, look for try-catch blocks in the documentation for the precise syntax.

The lifecycle of the killer is the same as before. After being hit 4 times the intermittent server will terminate the killer. Write a killer that is able to hit the intermittent server at least 4 times.

1.2 Trapping exits

Now you should have an attacking process that never crashes. Make a copy and prepare to expand its functionality, which will upgrade the killer to a *lord*.

In this exercise the lord must interact with the server via a number of processes that it spawns (called *knights*). Execute `srv:startK()` to run the server. The server will be registered locally via the name `knightServer`.

Intuition. The lord must spawn 15 knights that will fight for him. Knights fight the server 5 at a time; they are small extensions the process of Exercise 1.1. They have to be *registered* locally with a name that starts with `erlab_`. They attack and get attacked. They have 20 HP (like the killer of previous exercises). Every knight registers with the server and fights the server with the values it receives. When a knight dies, the lord is notified and, if present, another knight is sent to battle. When all knights are dead the lord registers itself and fights the server until it dies.

Implementation details. You have to create two types of processes: the knight and the lord, both are small extensions the process of Exercise 1.1, though knights are spawned by the lord. In this case it is advised to keep the implementation of the two processes as separate files. The lifecycle of knights and lord are analogous to that of the process of Exercise 1.1: they register with the server, they receive attack and defence values; they attack and receive attacks until they die. Small variations in their lifecycle's are listed below.

A knight registers with the server by sending a `registerKnight` message with its PID, its name and the lord's name. This will require a few modifications to the code of the process of Exercise 1.1. Knights must be *linked* to the lord process using the BIFs `spawn_link` or `link`. Linked processes monitor each other's status. Whenever a process terminates, it communicates the cause of the termination to all linked processes with an *exit signal*: a message of the form `{'EXIT',Id,Why}`.

When a knight process is terminated (after its HP are less or equal than 0), the lord must *trap* the event associated with the termination. Then the lord spawns another knight to replace the dead one, only if it still has live knights from its initial band of 15 knights. Otherwise it waits for all knights to die and then it registers itself with the server, behaving like the process of Exercise 1.1. Trapping *exit signals* is achieved by using the BIF `process_flag(trap_exit, true)` inside the code of the lord. This call will convert exit signals sent to the lord into messages of the form `{'EXIT',Id,Why}`, which can be handled in a `receive` block.

Additional messages to be sent and received have the following structures:

Message structure	Example
`{registerKnight, KnightPid, KnightName, MasterName}`	`{registerKnight, 3012, Gothmog, Morgoth}`

The UML sequence diagram in fig. 1 indicates the lifecycle of the first spawned knight and the lifecycle of the sixth one, which is intuitively spawned when the first knight dies.

1.3 Multiple local communication

Consider the process created in Exercise 1.1 (from now on the *killer*).

In this exercise you will create a killer and then spawn a number of them that will fight each other, under the surveillance of the server. To run the server, execute `srv:start0()`. The server will be registered locally via name `officialServer`.

Intuition. Up until now you created processes that fight an immortal server. It is time to spawn several copies of them and have them fight each other, the server will just act as a sort of referee. Instead of having a single known opponent (the server), killers now must look for other processes that look like them. Such processes are registered locally and have a name starting with `erLab_`. In order to spot such processes (from now on *enemies*), the killer can be helped by a *scout* process, whose code you do not have to write and can be found in the `scout.erl` module. The killer should keep track of all enemies and whether they are alive or not.

Live killers register themselves locally and register with the server in order to receive the attack and defence values. The server is used as a logger, keeping track of registering killers, attacks, defences and dead killers. When a killer is dead, it must inform the server, who is then in charge of sending this information to all other killers.

Description of the killer. The killer must send attack messages to all processes it receives names of from the *scout*. The killer should maintain a list of names that the *scout* communicated in order to know to whom to direct its attack messages. Attacking is done by sending the killer's PID, its name and the value of attack on atom `atk` to an atom contained in the list of names. Additionally, when attacking, a `atkLog` message must be sent to the server containing the killer's PID, its name, the target's name and the weapon power.

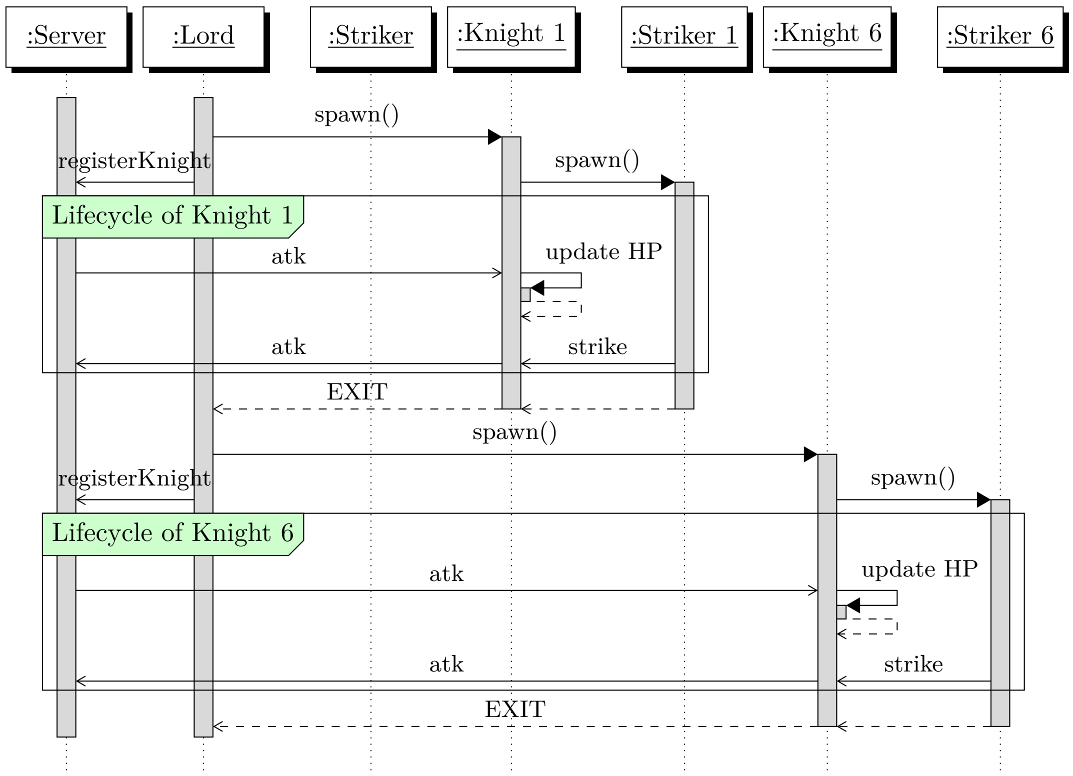

Figure 1: Sequence diagram for Exercise 1.2

The killer can be a target of the attacks of other killers; thus it also receives `atk` messages. The only difference with before is that it is not the server sending such messages, they come from other killers. When receiving an attack, the killer calculates is new amount of HP as described before. Additionally it sends a `dfnLog` message containing its PID, its name, the attacker's pid, the attacker's name, its defence value and the damage to the server. The server logs everything.

When a killer dies it communicates its PID and its name to the server on the atom `killLog`. The server propagates the name of the killed process to all players via a `killed` message (with a fake value for the expected `Pid`). Thus the killer can receive messages of the form {`killed, Pid, Name`}, meaning that process called `Name` is no longer running. In this case he should update the list of names and remove the terminated entry `Name`.

Scouting for enemies. The killer must spawn a process *scout* that will monitor the registered processes in the system *every 200 msec*, communicating the newcomers to the killer. The *scout* is contained in the file `scout.erl` and it can be started by executing the function `scout:scout(Name,Par)`, where `Name` is the name with which the scout must register itself and `Par` is the name of the killer.

The Scout sends these kind of messages to the killer:

Message structure	Example
{`addEnemy, ScoutName, EnemyName`}	{`addEnemy, Balrog, Fëanor`}

Such messages contain new enemy names. The killer must update the list of enemy names based on what names the *scout* communicates to him. The *scout* will communicate only newly registered processes.

Unfortunately, when a killer dies, the server communicates this event only to other killers and not to their *scouts*. Killed enemies are to be removed from the list of enemies of a killer. When a killer is informed by the server that a process has died, it must communicate it to the *scout* so that the scout can update the knowledge of living enemies. The messages the killer sends to the *scout* are:

Message structure	Example
`{remove, ToRem}`	`{remove, Fëanor}`

When receiving such messages the *scout* knows that he should not communicate the name of the dead process anymore to the killer.

Important. Write a main function that spawns at least 3 of these killers locally. Look at the logs to understand how they fight each other. Test with more processes and add delays between the spawning of different fighting processes.

Additional messages to be sent and received have the following structures:

Message structure	Example
`{atkLog, Pid, Name, TargetName, WeaponPower}`	`{atkLog, 6798, Gothmog, Fëanor, 10}`
`{dfnLog, Pid, Name, AttackerName, DefenseVal, Damage}`	`{dfnLog, 6798, Fëanor, Gothmog, 4, 6}`
`{killLog, Pid, Name}`	`{killLog, 6798, Fëanor}`

The UML sequence diagram in Figure 2 indicates the lifecycle of two killers, where the first one kills the second one and then notifies its scout.

2 Further experience

This exercise will combine the processes created in Exercise 1.3 and Exercise 1.2. You shall create a process (from now on the *lord*) that can spawn knights and use a scout to detect other processes.

In this exercise you will spawn a number of killers as in Exercise 1.3 that will fight each other via knights as in Exercise 1.2 under the surveillance of the server. To run the server, execute: `srv:startM()`. The server will be registered locally via the name `maelstromServer`.

Firstly, create a function for the lord. Expand the process of Exercise 1.3 in order for it to spawn knights as in Exercise 1.2. The lord communicates with the scout and receives a list of enemies. Among such enemies there are also the knights the lord created, the lord should remove its knights from the list of enemies. Then the lord shall communicate a target to all knights so that they can attack an enemy. Create an message type for this communication as you prefer. You are free to implement any kind of enemy distribution, you can send all the knights to the same target or spread them. Remember that there can be only 5 knights at a time of the total 15.

Once the template of the lord is created, start the server, spawn a number of lords and have them fighting each other. Analyse the logs to understand what is going on or going wrong.

Good luck.

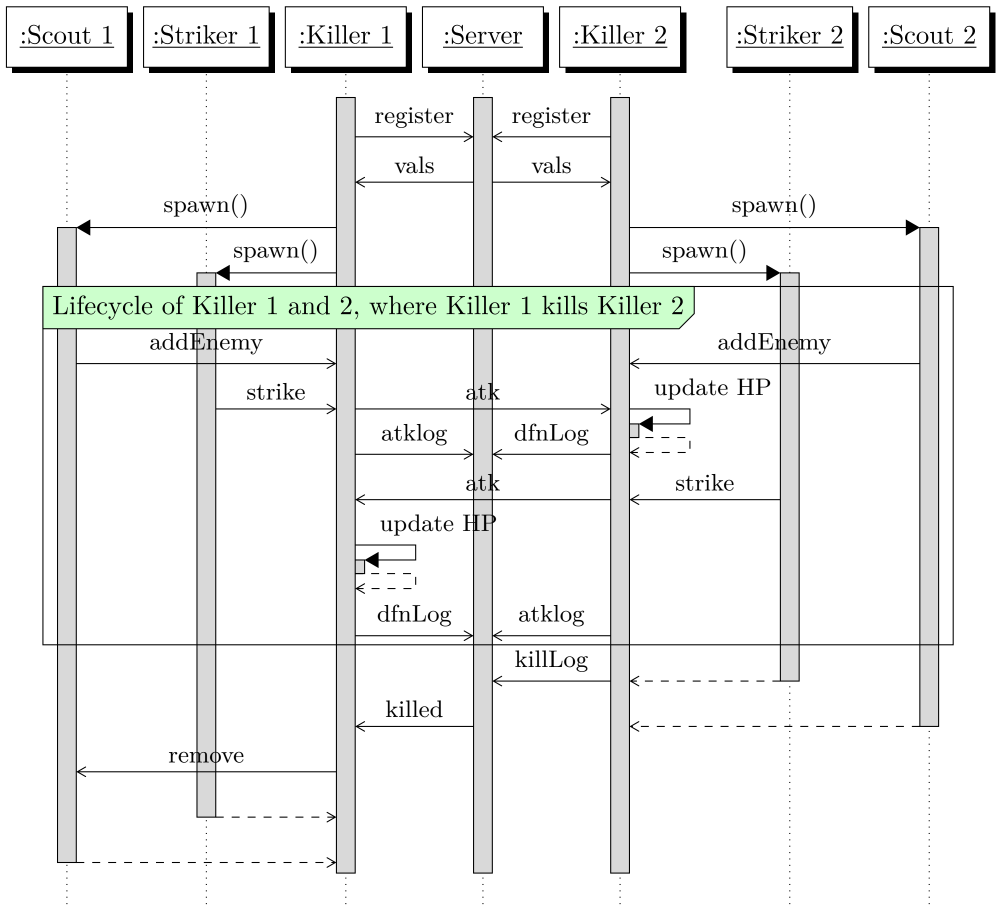

Figure 2: Sequence diagram for Exercise 1.3