# Concurrency Oriented Programming—Exercises

## Running Erlang

You can start an Erlang interpreter using the command ”erl”. You can also install Erlang on your own computer: downloads are available from erlang.org. Whether you use the version installed at Chalmers, or install your own, you will need the Erlang documentation. This is available at <http://erlang.org/doc/> (and is also installed along with the Erlang system when you download a copy to your own computer).

## Building a logger

When an Erlang process crashes, it may be hard to know what the process was doing before the crash. In this exercise, we will build a simple logger that enables processes to log events as they run, and have the last logged event automatically printed if the process later crashes.

The API you should implement consists of three functions:

* logger:start(), which starts the logger process,
* logger:stop(), which stops it,
* logger:log(Event), which sends the Event to the logger, and informs it that the sending process should be monitored.

For example, the following program uses the logger to report the exit reasons of two processes:

test() ->

spawn(fun() ->

logger:log(starting\_1),

Pid = spawn\_link(fun() ->

logger:log(starting\_2),

timer:sleep(1000),

logger:log(stopping\_2),

1/0

end),

logger:log(started\_2),

timer:sleep(100),

logger:log(about\_to\_kill\_2),

exit(Pid,reason),

timer:sleep(100),

logger:log(stopping\_1),

exit(done)

end).

The output when this program is run should be something like this:

25> logger:test().

<0.100.0>

Exited: <0.101.0> with reason reason

Last message: starting\_2

Exited: <0.100.0> with reason reason

Last message: about\_to\_kill\_2

(Try removing the calls to sleep, and see the effect on the output).

## Fault-tolerance by Replication

Sooner or later, every computer crashes. Fault-tolerant systems should continue to work *even though* the computers they run on crash. The only way to achieve this is for critical servers to be replicated, running on *more than one* computer. In this exercise, you will develop a simple replicated server, although, for simplicity, we will run both copies of the server in the same Erlang node.

The intention is that you should develop an enhanced version of the simple generic server I developed in the lecture, which automatically starts *two* servers rather than one. One of them will be the *main* server, registered under the specified name, and this will take care of the requests that clients make. The other will be a *backup* server, which also processes the requests, but does not send replies to the clients. If the main server should crash, then the backup server steps in and takes over, becoming the main server and spawning a new backup server. Of course, the main server should also monitor the backup server, and restart it too if necessary.

The effect should be that either server may crash without any effect on the behaviour that clients see (provided no requests are made during the short interval between the main server crashing, and the backup server taking over its work).

You should achieve this *only* by changes in the generic server module—the callback modules ought to be entirely unaffected by this extension. You can start from the module in server1.erl in the course handouts.

## Transactional Servers

In the lecture, we saw how to build a generic server with *transactional semantics*, in that crashing requests simply leave the server state unchanged. More generally, we might want to perform a *sequence* of requests to the server in one “transaction”. In general, a transaction is a sequence of requests which *appear to be atomic* to other processes. Thus, if we start a transaction, perform a sequence of requests, and then end the transaction, then other clients of the same server should only see server states before the transaction starts, or after it ends—no intermediate state should ever be visible to another client.

One way to achieve this would be to “lock the server”, and simply queue up requests from other clients until the entire transaction is over. But since a transaction may run for quite a long time, then this could restrict concurrency dramatically. Instead, transactions are usually implemented using *optimistic concurrency*—which mean that two or more are allowed to proceed in parallel, but at the end of a transaction, a check is made to see whether or not any interference actually occurred, and if so, then the transaction is *aborted*.

While a transaction is running, then its effects must not be visible to other clients, which means that the server must copy the server state when the transaction starts, and then perform the requests that make up the transaction on this copy, instead of on the main server state. In this way, other clients will see no state changes while the transaction is running.

When a transaction ends, then the copied state cannot simply replace the main state, because other operations may have changed the main state in the meantime. Instead, we must *apply the requests in the transaction* to the main state at this point—which we do atomically, thus guaranteeing that other clients see the state either before or after the entire transaction. Thus the server needs to record the requests that make up a transaction, so that they can be replayed when the transaction ends.

There is a problem, though. If another client has changed the server state while a transaction was running, then it’s possible that replaying the requests will generate *different* replies from those that have already been sent to the transaction client. If this happens, then the transaction cannot be completed, because the transaction client has already seen replies that differ from those that would be generated by performing the transaction atomically—so there is no way to make the transaction as a whole appear to be atomic. Instead, the transaction is aborted, and the client is informed. Usually, clients perform transactions in a loop, so that if the transaction aborts, then it is just repeated—perhaps after a delay.

Your task is to modify the generic server in server1.erl to support two new requests: start\_transaction, and end\_transaction. The first, start\_transaction, indicates that the calling client is starting a transaction, and subsequent requests from the same client pid should be processed by the server as part of this transaction. When the client eventually sends an end\_transaction request, then the transaction should be applied atomically to the main server state, provided there are no conflicts, and a Boolean returned to indicate whether or not the transaction succeeded.

Once again, your changes should be made *only* to the generic server code—the callback modules should be entirely unaffected. In this way, transactions can be added to any server at all, without changing the callback code in the slightest.

### Note

A very useful extension is to allow transactions to span over *multiple* servers—so that a client can, for example, withdraw money from one bank account, and deposit it into another, all as part of the same transaction, to guarantee that either both effects happen, or neither does. This can be implemented via a *two-phase commit*, where the client first asks each server whether it can commit the transaction, then if all reply positively, instructs each server to actually do so. If any server cannot complete the transaction, then the client instructs all of them to abort it. Why not implement this too, if you have time to spare?