Lab D: Distributed Erlang and Map-Reduce

The goal of this lab is to make the naïve map-reduce implementation presented in the lecture, a *little* less naïve. Specifically, we will make it run on multiple Erlang nodes, balance the load between them, and begin to make the code fault-tolerant.

Erlang resources

You will probably need to consult the Erlang documentation during this exercise. You can find the complete documentation here: http://www.erlang.org/doc/. To find documentation of a particular module, use the list of modules here: http://www.erlang.org/doc/man_index.html. Note that the Windows installer also installs the documentation locally, so if you are using Windows then you can just open the documentation via a link in the Start menu.

Connecting multiple Erlang nodes

The first step is to set up a network of connected Erlang nodes to play with. This can be done in two ways:

Running multiple Erlang nodes on one machine

Start several terminal windows/Windows cmd windows, and in each one start a named Erlang shell. Do this using a command such as

```
erl –sname foo
```

on Linux or the Mac, and

```
werl –sname foo
```

on Windows. (The Windows version starts Erlang in its own window, with some useful menus). The prompt displayed by the Erlang shell will show you what each Erlang node you created is called. For example, on my machine the prompt is

```
(baz@JohnsTablet2012)1>
```

This tells me that the node I created is called **`baz@JohnsTablet2012`** (an Erlang atom).

Running Erlang nodes on multiple machines

It's more fun using several machines. The procedure is the same as above, but first you *must* ensure that all machines use the same cookie. Edit the file .erlang.cookie in your home directory on each machine, and place the same Erlang atom in each one. Then start Erlang nodes as above; as long as the machines are on the same network, then they should be able to find each other. In particular, machines in the labs at Chalmers ought to be able to find each other.

Connecting the nodes together

Your Erlang nodes are not yet connected… calling **`nodes()`** on any of them will return the empty list. To connect them, call

```
net_adm:ping(NodeB).
```

on NodeA (two of your node names). The result should be **`pong,`** and calling **`nodes()`** afterwards on either node should show you the other. Connect all your nodes in this way. Note that because Erlang builds a complete network, then you need only connect each node to *one* other node yourself.

Help! It doesn't work

- On multiple machines, check that the cookie really is the same on all the nodes. Call **`erlang:get_cookie()`** on each node to make sure.
- If NodeA can't connect to NodeB, try connecting NodeB to NodeA. Sometimes that helps!
- Perhaps one or more of your machines requires a login before the network connection can be established. In a Windows network, try visiting the Shared Folder on each machine from the others—this may prompt for a password, and once you give the password then Erlang will also be able to connect.

Remote Procedure Calls

We'll start by making remote procedure calls to other nodes. We'll call io:format, which is Erlang's version of printf. Try

```
rpc:call(OtherNode,io,format,[“hello”]).
```

You will find "hello" printed on your *own* node! Erlang redirects the output of processes spawned on other nodes back to the original spawning node—so io:format really did run on the other node, but its output was returned to the first one. To force output on the node where io:format runs, we also supply an explicit destination for the output. Try

```
rpc:call(OtherNode,io,format,[user,”hello”,[]]).
```

(where the last argument is the list of values for escapes like ~p in the string… since "hello" contains no escapes, then we pass the empty list). Make sure that the output really does appear on the correct node.

Compiling and loading

Loading code on other nodes is very simple. Write a simple module containing this function:

```
-module(foo).
-compile(export_all).
foo() -> io:format(user,”hello”,[]).
```

Now you can compile this module in the shell via

```
c(foo).
```

and you can then load it onto all your nodes via the command

```
nl(foo).
```

Try using **`rpc:call`** to call **`foo:foo`** on each node, checking that the output appears on the correct node.

Naïve Map-Reduce

On the course web page you will find the source code of three of the modules presented in the lecture on map-reduce: a very simple map-reduce implementation on one node (both sequential and parallel), and two clients—a web crawler and a page rank calculator. Compile these modules, and ensure that you can crawl a part of the web.

```
crawl("http://www.cse.chalmers.se/",3).
```

You will need to start Erlang's http client first, using `inets:start().`

The page rank calculator uses the information collected by the web crawler, but it assumes that the output of the web crawler has been saved in a *dets* file—a file that contains a set of key-value pairs. You will need to use dets to do this lab. You can find the documentation here (http://www.erlang.org/doc/man/dets.html) –and there is quite a lot of it—but you will only need a few functions from this module.

- `dets:open_file`—see code below for usage
- `dets:insert`—which inserts a list of key-value pairs into the file
- `dets:lookup`—which returns a list of all the key-value pairs with a given key

Save the results from the web crawler in a dets file called web.dat, and check that the page ranking algorithm works. Then copy web.dat onto all your nodes—this will enable you to distribute the page-rank computation across your network. You should collect 40-100MB of web data so that the page-ranking algorithm takes appreciable time to run.

Distributing Map-Reduce

Modify the parallel map-reduce implementation so that it spawns worker processes on all of your nodes. Measure the performance of the page-ranking algorithm with the original parallel version, and your new distributed version.

Load-balancing Map-Reduce

Of course it is not really sensible to spawn all the worker processes at the same time. Instead, we should start enough workers to keep all the nodes busy, then send each node new work as it completes its previous job. Write a *worker pool* function which, given a list of 0-ary functions, returns a list of their results, distributing the work across the connected nodes in this way. That is, *semantically* `worker_pool(Funs) -> [Fun() || Fun <- Funs]`, but the implementation should make use of all the nodes in your network. A good approach is to start several worker processes on each node, each of which keeps requesting a new function to call, then calling it and returning its result to the master, until no more work remains to be done. Modify the map-reduce implementation again to make use of your worker pool in both the map and the reduce phases. Measure the performance of page ranking with your new distributed map-reduce… is it faster?

Fault-tolerant Map-Reduce

Enhance your worker-pool to monitor the state of the worker processes, so that if a worker should die, then its work is reassigned to a new worker. Test your fault tolerance by killing one of your Erlang nodes (not the master) while the page-ranking algorithm is running. It should complete, with the same results, despite the failure.

Hand ins

You should submit the code of the three versions of map-reduce described above, together with your performance measurements. Describe your set-up: were you running on one machine or several, how much web data were you searching? What conclusions would you draw from this exercise?

The deadline is 23:59 on Monday, 16th May. The final deadline is a week later.

More

A full map-reduce implementation does a lot more than this, of course. The next step would be to avoid sending all the data via the master—the results of each mapper should be sent *directly* to the right reducer… although this introduces a lot more complexity. Something to experiment with later, perhaps?