ELIoT - ErLang for the Internet of Things

Alessandro Sivieri

November 9, 2012

1 Introduction

ELIoT is a framework for developing applications for the so-called "Internet of Things" scenarios using Erlang.

It is constituted by three parts:

- a specialized version of the Erlang Virtual Machine
- a set of libraries
- a simulator.

1.1 The Virtual Machine

The VM has several important differences with respect to the original VM.

The main difference is the distribution protocol: while Erlang uses a TCP-based protocol for communication between nodes (instances of the VM running on different machines), the ELIoT protcol is UDP-based and does not have any concept of "session": a standard port (4369/UDP) is opened when a node is launched, and it waits for incoming messages for that node; each Erlang process communicating with the outside world will send a UDP message towards a specific node. There is no concept of acknowledgements (except in some particular cases, as described later), and in general a UDP message may be lost during the communication, without either party knowing this fact. As a consequence of this change in protocol, each machine/device can only have one single instance of the VM running, and the list of neighbors maintained by the library is not reliable anymore.

Deriving from the previous change, we introduced a new operator for unicast communication and a new mode for broadcasting messages in a network. In unicast, if the "bang" operator ("!") is used, then the distribution protocol tries to achieve some degree of reliability: the message is sent and an acknowledgement is expected by the other party; if not received after a certain timeout, the message is resent. After a certain number of tentatives, the driver gives up and sends a NACK message to the application level. If the ACK is not necessary/wanted, then a completely unreliable message can be sent by using the "tilde" operator

("~", not existing in Erlang): if used, the driver will simply send the message using UDP and forget about it.

In Erlang, broadcasting is usually performed by sending n single unicasts to the neighbors (using the *erlang:nodes/0* function); as stated before, in our implementation the neighbors list is not reliable anymore, so broadcasting a message means sending it to the (fictious) node called "all", which is interpreted by the driver as a message to be broadcasted to the entire subnetwork (i.e., if the local address is 192.168.1.23, the message is sent to 192.168.1.255).

A new operation called "export" is introduced (*erlang:export/1*), which has to be invoked for allowing a process to receive messages from the network: while in Erlang usually each process can receive messages from the outside world (given that its name or its process identifier has to be known by any other party in the communication), in ELIoT a process has to be "exported" (by name or by pid), and from that moment on it will receive messages coming from the network. Other operations allow a process to be "unexported" and to obtain a list of all locally exported processes.

1.2 The library

The ELIoT library contains several parts:

- the UDP driver that is used as the new distribution protocol (note that the ELIoT VM is not able to use the default TCP driver anymore)
- the main API (the *eliot_ api* module), which allows a node to save some data so that it will be available to all the processes running in the same node, and spawn other processes in the whole subnetwork (or in a single node), and other utility functions
- several algorithms used in distributed systems (i.e., CTP, trickle)
- the simulator library (see the next subsection for details)
- some low level libraries used for interfacing with hardware (i.e., GPIO, WiFi modules).

There are also a few examples for testing these algorithms, in the subfolder "examples", and other tools used for simulation.

1.3 The simulator

The simulator allows two different modes:

- full simulation: a set of nodes is created and simulated inside a single machine (VM instance)
- mixed simulation: a set of nodes is simulated in a single machine, while some other nodes are executed on different devices: this allows the code to be tested also on real devices, and at the same time the size of the network

can be greater than the number of devices that can be used (both simulated and real devices can communicate with each other in this scenario).

The code executed in simulations is exactly the same to be executed on real devices: it has only to be recompiled and then executed in the simulator; of course, some operations can be introduced so that in a full simulation some data can be injected in the network (i.e., in a dissemination scenario, some new data can be introduced on a single node, so that the dissemination of that data can be started).

2 Installation

2.1 Installing the VM

Before installing the VM, some dependencies need to be installed in the system; in particular, for Ubuntu the following packages have to be installed:

build-essential automake wx-common wx2.8-headers libwxgtk2.8-dev unixodbc-dev libncurses5-dev fop flex bison perl5 default-jdk libsctp-dev libssl-dev

The ELIoT VM git repository can be cloned at the URL `gitosis@sola0.dei.polimi.it:erlang.git`, and can be downloaded (for authorized users only) from there; it can be consequently compiled by executing the following commands from a shell:

./otp_build autoconf

./configure --disable-hipe --prefix=/some/path

make

make install

It could be useful if the path specified during configuration is different from the system ones (i.e., */usr/*, */usr/local*), so that the ELIoT installation does not interfere with the standard one that can be installed using the package manager of the chosen Linux distribution.

2.2 Compiling the framework

The ELIoT framework can be cloned at the URL `gitosis@sola0.dei.polimi.it:eliot.git`, and it can be compiled by executing *make* from the command line (the ELIoT VM "bin" subdirectory should be in PATH, and this can be done by executing

export PATH=/some/path/bin:$PATH

from the command line). Before compiling the code, there are two files that have to be edited by hand (at least for now):

- include/eliot.hrl
- deps/udp_dist/include/eliot.hrl

They are actually the same file duplicated in different locations; in both files, the constant value called *interface* has to be changed to reflect the name of the network interface in use (i.e., *eth0/eth1* in case of a wired interface, *wlan0* in case of a wireless interface; refer to the *ifconfig* command to know the currently used interface name). This is important, because this information is used by both the ELIoT driver and the framework for effective communication.

There is also a third file:

- deps/udp_dist/src/udp_server.erl

This file has, at line 44, a path specifying where the interpreter can find the new UDP driver that has been developed (which is located under the *priv* directory), and it has to reflect the full path where Eliot has been downloaded.

2.3 Executing the interpreter

The interpreter can be executed from the main ELIoT directory (the same of the previous subsection) by launching the following commands from a shell:

```
shell> /some/path/bin/erl -pa ebin deps/*/ebin -proto_dist udp
-no_epmd
erl> udp:start().
```

The first is executed in the shell, and when the Erlang prompt is ready, the second line has to be executed; this will start the distribution protocol (some debug messages can still be shown from time to time, and they can be ignored usually). At this point, the interpreter can be used as usual and also for communicating to other nodes executed on different machines. If the network distribution is not used (for example for executing a local application), then the *udp:start()* command can be ignored.

2.3.1 Executing an example (on a device)

For executing one of the examples, it is necessary to execute *make* again in that directory (checking that the *deps* directory contains a link to the main ELIoT directory); at this point, the shell script *start.sh* contained in each directory will start the algorithm.

2.3.2 Executing an example (on the simulator)

The *make* command has to be executed with the argument *simulation*: this will recompile all the code (the example and the framework itself), and at this point the same shell script as before can be executed with the same argument, and the simulation nodes will be launched (and may produce some output, depending

on the application being simulated); at this point, there are some operations that can be used to simulate data collection/dissemination or other operations, please refer to each application to know more about it.