Calvin: A Virtual Machine for Cal

Some quick notes

Patrik Persson, Ericsson AB (patrik.j.persson@ericsson.com)

March 24, 2014

Contents

1 Introduction

Calvin is a Cal runtime system that allows for dynamic loading of actors. The idea is to have Calvin running continously, and load/unload actors and networks as needed.

These instructions have been verified on my Ubuntu VM, and should probably work on any recent Linux machine. Support for Mac OS X is experimental. This document is by no means complete; work on it has just begun.

1.1 Components involved

Running an application requires the following:

- Calvin
- one or more actors, in the form of hot-loadable .so files
- a network description, typically in CalvinScript format

1.2 How to build Calvin and actors

First, build Calvin in the most straight-forward way:

```
make
```

Build example actors, as separate .so files (Linux) or .bundle files (Mac OS X). The runtime makefile has a rule for building such files. On a Linux machine, do this:

```
cd examples/loadableActors
make -f ../../Makefile \
  Example__m1.so Example__m2.so \
  Example__m3.so Example__m4.so
```

Or, on a Mac:

```
cd examples/loadableActors
make -f ../../Makefile \
  Example__m1.bundle Example__m2.bundle \
  Example__m3.bundle Example__m4.bundle
```

1.3 How to run it

For the 'loadableActor' example above, the network is described in Example.cs. Assuming that 'examples/loadableActors' is still the current working directory:

```
../../calvin Example.cs
```

This line loads Calvin, loads four actors, and finally loads a network. The network is then executed. You should now have four files named outdata1..outdata4.txt in your directory.

The extension '.cs' stands for 'Calvin script'. This way of describing the network is described in Section 2.

1.4 Considerations for code generation

Each generated actor has a global variable 'klass' referring to the actor class. This value previously had a unique name for each actor. Diff output:

```
-ActorClass ActorClass_Example_m1 = INIT_ActorClass(
+ActorClass klass = INIT_ActorClass(
```

Until the compiler generates such code, here's a command line that you may find useful in this conversion:

```
sed -ie 's/ActorClass.*INIT_ActorClass/ActorClass klass=INIT_ActorClass/g' *.c
```

2 Calvin scripts: executable network descriptions

To be able to deploy applications in an incremental and distributed fashion, a simple network description language has been designed. It is intended to be used in (at least) the following three ways:

- In a network description file. In this case, such a .cs file contains a sequence of commands that build and execute a Cal network.
- As a plaintext protocol between devices (like IETF protocols such as POP). In this case, commands are issued by a remote peer, for example, to establish network connections.
- Line-by-line execution, to allow for interactive operation, for experimentation and system maintenance.

Calvin thus interprets CalvinScript commands and executes the resulting actor network accordingly.

2.1 Command-line arguments

Executing an entire network description file was demonstrated in 1. Line-by-line execution is invoked by the -i option, as shown in the following example:

```
# ./calvin -i
OK calvin 1.0
% load ./Example__m1.so
OK Example_m1
% classes
OK art_Sink_bin art_Sink_real art_Sink_txt ... Example_m1
% quit
OK bye
```

Calvin can also be started as a server on a given TCP port using the -s option, for example as

```
# ./calvin -s 9000
```

The server can then be remotely accessed using, e.g., telnet:

```
# telnet <ip address> 9000
OK calvin 1.0
% quit
OK bye
```

Command line parameters are executed in order, so the following command line

```
# ./calvin -s 9000 -i
```

spawns a server thread, then immediately launches an interactive session. This is probably what you want; it allows you to monitor Calvin from the same console. In contrast,

```
# ./calvin -i -s 9000
```

launches an interactive session, waits for it to terminate (by a QUIT command), *then* spawns a server thread. This is probably *not* what you want.

2.2 Command reference

Commands in Calvin are used for the following:

- Loading classes
- Creating actor instances of those classes
- Connecting actor instances
- Executing actor networks

The response to a command, as shown either interactively or via a network socket, is a one-line status line beginning with either 'OK' or 'ERROR'. After this indicator follows more information; usually an informal status message, but in some cases (such as CREATE or LISTEN) also important information.

Commands are not case sensitive; identifiers are, however.

Comments start with # (the hash character) and extend to the end of the line. Empty lines are allowed and constitute no-ops.

The actor network generally executes asynchronously to the command parser.

ACTORS

Syntax:

```
ACTORS
```

Lists actor instances.

Calvin will respond with a single line, starting with 'OK', followed by a number of instance names separated by whitespace.

ADDRESS

Syntax:

```
ADDRESS
```

List the IP-address(es) of the host. (?)

CLASSES

Syntax:

```
CLASSES
```

Lists actor classes.

Calvin will respond with a single line, starting with 'OK', followed by a number of class names separated by whitespace.

CONNECT

Syntax:

```
CONNECT <source actor>.<source port> <destination actor>.<destination port>
CONNECT <source actor>.<source port> <remote IP address>:<remote port>
```

Connects an output port of one actor to an input of another.

The destination is either a local actor, or the port of a remote host which has previously executed a LISTEN command. The remote port is then the port provided as output by that LISTEN command.

DESTROY

Syntax:

```
DESTROY <actor1> [<actor2> ... <actorN>]
```

Destroys the listed actor(s)

For each listed actor all inputs are disconnected from the producers and all outputs are disconnected from the consumers, freeing output buffers, before calling the destructor of the actor.

ENABLE

Syntax:

```
ENABLE <actor1> [<actor2> ... <actorN>]
```

Schedules one or more previously created actors for execution.

Until this is done, the actors will not be executed.

JOIN

Syntax:

```
JOIN
```

Blocks the command parser until no more actors can fire.

This command provides a mechanism for detecting the termination of an application. If the network does not go idle, then this command will not terminate.

(This command currently requires the entire network to go idle, which is rather crude. When executing multiple applications concurrently on the same node, it may even be useless. Joining a specific actor is currently not supported, but may be in the future.)

LISTEN

Syntax:

```
LISTEN <actor>.<port>
```

Initiate a remote connection.

Specifies that the indicated port is to receive tokens from a remote source, rather than a local one. Upon success, the status line will be 'OK' followed by whitespace and the numeric IP port where tokens are expected. This IP port is to be used by the remote peer's CONNECT command.

LOAD

Syntax:

```
LOAD <file name>
```

Loads a compiled actor from the indicated binary file.

It is not always possible to infer the actual name of the class from the file name. For this reason, the status line will provide this information: it consists of the word 'OK', followed by whitespace and the name of the class.

NEW

Syntax:

```
NEW <class name> <instance name> [<param1>=<value1> ... <paramN>=<valueN>]
```

Create a new instance of an actor class.

Parameter values, if applicable, are provided on the same line. After this command has been given, the actor instance is not yet scheduled for execution: the command ENABLE must be given for this to be done.

QUIT

Syntax:

```
QUIT
```

Terminates a session.

SHOW

Syntax:

```
SHOW <actor>
```

Show a description of the actor.

3 Dynamic deployment of distributed applications using Python

CalvinScript can be viewed as a simple protocol, where the client sends a single-line request and the server (Calvin) responds with a single-line response. A client for this protocol has been implemented in Python. This client allows Calvin nodes, actor classes, actor instances, and ports to be represented as Python objects.

A simple distributed application is given in the following example:

```
import calvin

n1         = calvin.Node("localhost", 9000, True)
n2         = calvin.Node("localhost", 9001, True)

sensor     = n1.new("art_Source_txt", "sensor", fileName="in.txt")
actuator   = n1.new("art_Sink_txt", "actuator", fileName="pyout.txt")

comp_class = n2.load("./Example__m1.bundle")
comp       = n2.new(comp_class, "comp")

sensor.Out >> comp.In
comp.Out   >> actuator.In

sensor.enable()
actuator.enable()
comp.enable()
```

The example assumes two Calvin instances to be running on the local machine, on ports 9000 and 9001 respectively. Start these in separate terminal sessions using

```
calvin -s 9000 -i
```

and

```
calvin -s 9001 -i
```

respectively. Make sure the Calvin instance on port 9000 runs in the same directory as the .so (or .bundle) actor file; if necessary, compile it as outlined in QuickIntro. The application above can then be executed as (assuming it has the name 'example.py'):

```
python example.py
```

The Python operations on Node objects largely correspond to CalvinScript commands. However, the new() operation returns a Python object representing the remote actor, and this actor object in turn holds members representing the actor's ports. This allows connections to be described succinctly using the >> operator. Note that the connections in the example are of the remote variety; the Python client will then perform a LISTEN-CONNECT pair of commands to establish the connection. If the 'comp' actor were instead deployed on the same node as the sensor and the actuator, the connections would be local, and the >> operator would perform a local FIFO connection.

Also note that the Node::load() operation returns a string in the format to be used for Node::new(), as shown for 'comp_class' in the example.

4 Some notes on the implementation

Calvin is multithreaded, not (currently) to use multiple cores, but to permit concurrent access from an interactive session and one or more remote sessions. These notes describe the general ideas in the threading design. They are intended to help the reader understand the code.

The actor worker thread

All actors share a single worker thread implemented in actors-network.c. Contrary to some other Cal runtime systems, this thread does not terminate: rather, it goes idle when there is no work to do. Calvin is viewed as a resident service on a node, rather than a one-shot application.

- When all actors have been executed once, without a single firing, the worker thread will suspend itself.
- When an actor is enabled (using the ENABLE) command, a new connection is added (using the CONNECT command), or a socket is ready for sending/receiving, the worker thread will be activated.

Socket receivers/senders

Remote connections, that is, connections between actors residing on separate nodes, are implemented using dedicated proxy actors. These proxy actors forward tokens to/from sockets to/from local token FIFOs. The proxies are implemented in actors-socket.c.

Each such proxy has a dedicated, separate thread to send/receive tokens. These separate threads block on read()/write() calls, pass tokens to/from the proxy, and notify the worker thread when there is work to do.

Synchronization

The actor network, handled in actors-network.c, is accessed from multiple threads, including both the main thread (running commands from the command line) and any threads associated with remotely connected clients. These threads are collectively referred to as 'the parser' in actors-network.c.

A simple, straight-forward approach to synchronization has been taken. Internal structures are generally built upon a doubly-linked list, described in dllist.h. The implementation of this list is intended to be thread-safe, and each list instance is associated with a lock. The locking is rather fine-grained, in that two parts of a list can be concurrently accessed.