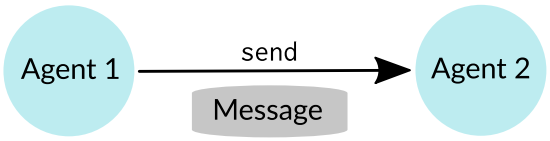
Erlangen: Introduction

Max Rottenkolber <max@mr.gy>

Sunday, 4 December 2016

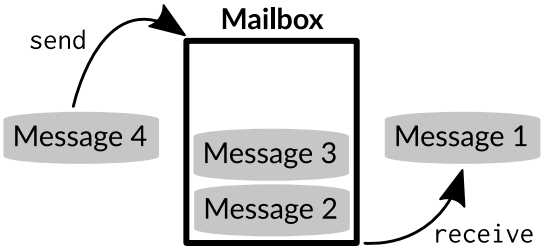
Erlangen (https://github.com/eugeneia/erlangen) brings distributed, asynchronous message passing to *Clozure Common Lisp* (http://ccl.clozure.com/). It orchestrates Clozure CL *processes* (native threads) using *message passing* (http://c2.com/cgi/wiki?MessagePassingConcurrency), and encourages fault tolerant software architectures using *supervison trees* (http://erlang.org/documentation/doc-4.9.1/doc/design_principles/sup_princ.html). It is also transparently distributed, all its features work seamlessly across IP networks. Thus, it can be used to build applications across multiple Clozure CL instances on different hosts. Erlangen borrows many ideas from *Erlang/OTP* (http://www.erlang.org/), hence the name. (Its a town!)

Clozure CL processes are comparatively heavy weight, preemptively scheduled operating system threads, as opposed to Erlang's cooperatively scheduled green threads. As such, processes are a scarce resource on Clozure CL. Erlangen is based on the assumption that even with only a limited capacity of concurrent processes, message passing and supervision trees are still feasible features. While Erlang's distribution features are sometimes overlooked, they are meant to be a main focus of Erlangen (but not of this article). Besides the overlap in features, terminology, and name, Erlangen and Erlang/OTP are completely unrelated, incompatible, and fundamentally different.

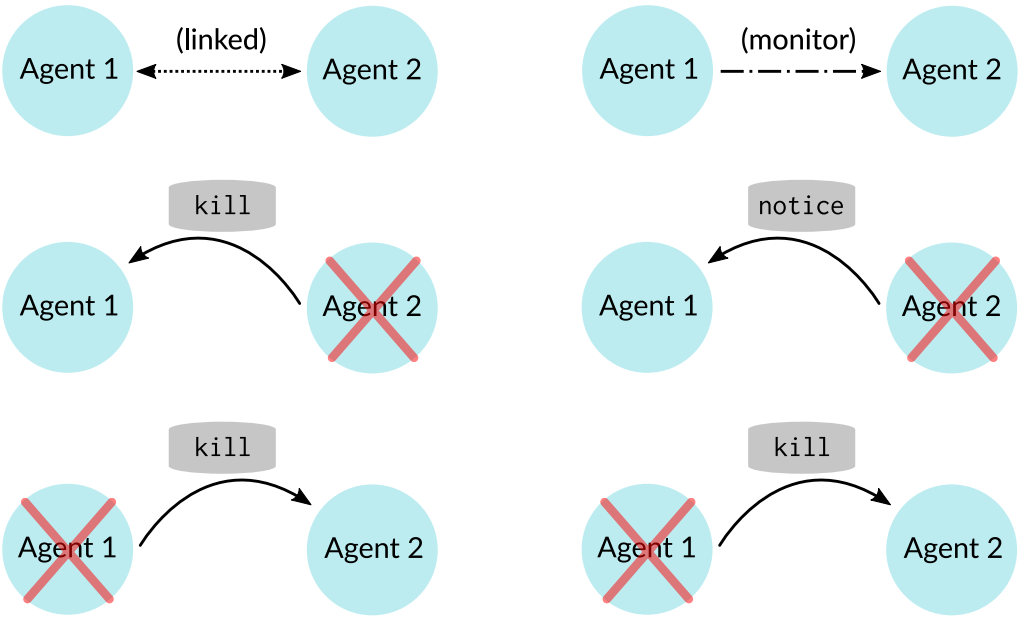


Erlangen equips each process with a *mailbox*, a list of *links*, and a list of *monitors*. These upgraded processes are called *agents*. Agents can *send* (http://mr.gy/software/erlangen/api.html#section-1-13) messages to the mailboxes of other agents, and *receive* (http://mr.gy/software/erlangen/api.html#section-1-9) the next message from their own mailbox. Message delivery can fail when the target agent exited, the destination mailbox is full (mailboxes are bounded FIFO queues), or due to network failure. As such, delivery of any message to the target agent's mailbox

is not guaranteed. Agents can supply a timeout argument to `receive`, which will then signal a `timeout` error if no message is received until the timeout is exceeded.



Agents can also *exit* (using *exit* (http://mr.gy/software/erlangen/api.html#section-1-6) or by returning normally), and send *kill messages* to other agents. When an agent receives a kill message it exits immediately, and the remaining messages in its mailbox are discarded. The links and monitors of an agent are references to other agents. When an agent exits it sends kill messages to all its links, and *exit notifications* to all its monitors. Agents can use *link* (http://mr.gy/software/erlangen/api.html#section-1-7) and *unlink* (http://mr.gy/software/erlangen/api.html#section-1-16) to manage links and monitors. Unlike regular messages, kill messages and exit notifications will never be lost in transit, and they are prioritized so that they will be received before any pending regular message. Unless the target agent exits before it receives the kill message or exit notification, they are guaranteed to be delivered, eventually.



Finally, agents can *spawn* (http://mr.gy/software/erlangen/api.html#section-1-14) new agents. `spawn` takes a nullary function for the agent's process to execute, and returns a reference to the newly created agent. This reference can then be used to communicate or link with the agent. Additionally, `spawn` can also link the caller and the new agent before the new agent's process starts. This is useful to avoid a basic race condition where the child agent exits before the parent could link with it.

That covers pretty much all of the core functionality, but Erlangen is an asynchronous programming framework for a synchronous programming language, and that poses some problems. Our processes are preemptively scheduled, and it is unsafe to stop them asynchronously. Erlangen agents must call `receive` periodically, or else they become unresponsive to kill messages. Specifically, blocking calls to input and output routines, such as found in the Common Lisp standard, can pose problems in some cases. For instance, imagine an agent that listens for user input on an interactive input stream.

```
(spawn 'read-char)
```

The resulting agent would block until a character is available in `*standard-input*`, and not respond to incoming messages, potentially indefinitely. This is a big no-go. One solution is to implement non-halting agents as event loops. Instead of blocking while waiting for a single event, they continuously poll for multiple events like pending input and messages. The *select* (http://mr.gy/software/erlangen/api.html#section-1-12) macro helps to implement such an event loop. At its heart is a pacing poll algorithm that takes naps when there is nothing to do, thereby trading a little bit of initial latency for saved processor cycles in times of low load. Given alternative, asynchronous input routines the pathological case from before can be avoided using `select`:

```
(spawn (lambda ()
         (select ((read-char-no-hang) (char)
                  char))))
```

This agent polls for new input using the non-blocking `read-char-no-hang` while also receiving pending messages. Since there is no `:receive` clause, all incoming messages are discarded (but kill messages will still cause the agent to exit). When `read-char-no-hang` returns a non-nil object as its first value, the clause's lambda list `(char)` is bound to the return values, the body of the clause is evaluated, and its result returned. In this case, `char` is returned,

and the agent exits subsequently with that value. To manage additional types of events, we would add more clauses to `select`.

To summarize, agents need to be programmed with concurrency in mind. Within the boundaries of the Erlangen framework, this will come naturally by composing programs using the provided message passing facilities. When faced with outside input, use of blocking routines may not be acceptable. Sadly, this affects many of the standard Common Lisp I/O routines, and Clozure CL's socket stream extension is only of limited help, too. Asynchronous I/O libraries like *IOLib* (https://common-lisp.net/project/iolib/) provide a way out.

If you are interested in this framework, you are welcome to create an issue on *GitHub* (https://github.com/eugeneia/erlangen/issues), or shoot a mail to me at *max@mr.gy* (mailto:max@mr.gy). Let me know what you think!