Viewpoints Research Institute

Supporting Actors in COLA

Michael FIG

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2009-013

Supporting Actors in COLA

Michael FIG

michael@fig.org

2009–10–13

Contents:

1 Introduction

The Actor model is an easy-to-understand concurrency model that is gaining popularity as language support for it becomes more common. It is a natural choice for the foundational concurrency mechanism that applications can use directly or as a building block for more elaborate concurrency models. Its simplicity does not preclude dynamic modification of the topology of the concurrency system on the grounds of complexity.

The COLA Kernel Abstraction [Piumarta09] lets us implement message passing semantics in terms of composable layers. Every message is either a "downcall" which needs to be resolved by an underlying kernel, or an "upcall," where the sender has a direct reference to the destination mailbox and knows the exact procedure by which to correctly add the message. Transaction protocols among actors can implement more elaborate synchronised algorithms using these underlying mechanisms.

Ad-hoc message systems can be assimilated within these semantics. Asynchronous operating system services (windowing systems, signal delivery, network communications, coprocessor or thread pool invocations, etc.) fit naturally into this general mechanism.

The remainder of this memo describes an abstraction in which application, library, and system code remains agnostic to the specific concurrency models selected for itself or for its peers. This allows us to implement concurrency in COLA once and have it be useful in many different contexts.

2 Actors

An actor is an computational entity that can:

- send messages to other actors,

- create other actors, and
- designate the behaviour to be used for the next received message.

We extend this definition with features inspired by the Erlang programming language, so that actors can also:

- selectively scan their mailbox (leaving some messages unactivated),
- specify another actor to be monitored and receive a failure message if that other actor becomes unresponsive, and
- designate a timeout for sends and receives, to aid in recovery from actors that stop responding to application protocols.

3 Messaging Specification

The implementation is organised into layers, each of which is optional according to the application's requirements. Additional layers can be inserted into the stack as needed.

These are currently:

fibre
: The current COLA context, either a green thread or a coroutine.

thread
: An operating system thread.

process
: An operating system process.

platform
: A physical or virtual machine on which the operating system runs.

3.1 Schedulers

The `install-`*`LAYER`*`-kernel` functions (where *`LAYER`* is the name of a layer) inserts a kernel on the current context that provides scheduling and messaging services between the named entities in the current context.

```
(install-fibre-kernel)
(install-thread-kernel)
(install-process-kernel)
(install-platform-kernel)
```

After one or more of these kernel layers is installed, the following functions become available to the current context:

`(fork` *`closure [options...]`*`)`
: creates a peer of the topmost kernel (such as a new fibre) running the *closure* and returns a reference to its actor.

`(create-`*`layer closure [options...]`*`)`
: creates a peer of the specified kernel *layer* and returns the corresponding reference.

`(connect-`*`layer [options...]`*`)`
: Returns a reference to an existing entity.

The *`options...`* communicate parameters to the scheduler that could inclued timeslice quanta, priorities, and so on.

3.2 Actors

Messages are exchanged with an actor via its reference.

`(send` *`timeout destination message [references...]`*`)`
: Asynchronously adds *message* to the *destination* actor's mailbox along with optional *reference*s to other actors. The *timeout* is the duration in seconds (nil means infinite) that the message system should continue to try placing one copy of the message in *destination*'s mailbox. `send` returns `ack` if the message was successfully placed in *destination*'s mailbox, `down` if *destination* is known to have crashed, a more general error object describing the status of a failed send, or `timeout` if *timeout* expired before any other condition was reached.

`(receive` *`timeout match-function`*`)`
: Executes `(`*`match-function`*` message references...)` for each message in the current actor's mailbox. If the result is non-nil the message is removed from the mailbox and the result returned to the caller of `receive`. If no messages are received the process repeats for each newly-arrived message until a match is found. If no match is found within *timeout* seconds, `receive` returns `nil`.

`(current-actor)`
: Returns a reference to the current actor.

`(actor-kernel` *`n`*`)`
: Returns a reference to the kernel *n* layers below the current actor.

`(actor-`*`layer`*`-kernel)`
: Returns a reference to the kernel at the named *layer* under the current actor.

4 Security

Untrusted senders are permitted to send messages to trusted receivers. This section describes the related security issues, and how their solutions determine implementation rules.

1. Direct pointers to the receiver's mailbox allow the sender to corrupt the receiver.

   Untrusted code must be written in an object-capability form, so that the kernel controls all privileged operations. The kernel can wrap actor addresses in kernel handles, so that the untrusted code can use them without having access to their internal representation.

2. Mutable data structures in messages would allow the sender to modify a message after it is received and validated but before some trusted operation is performed on it.

   Message bodies must be copied by the kernel into the receiver mailbox so that they are no longer referenced by the caller. The receiver must only interpret message data as a reference if it is known to point to an object that is either immutable or untrusted.

3. Senders can cause a denial-of-service by filling receiver mailboxes faster than can be processed.

   It is the responsibility of kernels to throttle their senders by checking the return value of the mailbox post function. The COLA kernel's scheduler should penelise the sender and all its children each time a send "overflows" a mailbox.

4. Senders can cause a denial-of-service by sending huge messages.

   Receiver mailboxes should have a dynamically configurable maximum message size. Oversized messages are handled similarly to "overflow", but a special "oversize" condition should be signalled.

5 Optimisation

Some optimisation can be performed by avoiding copying messages when actors trust their senders and a shared memory space is available. Flow control need not be implemented, if memory is plentiful.

6 Further Work

Implementation work is still ongoing. This research memo documents the current intentions and solicits comments on improvements to the design.

Implementation techniques can be investigated and borrowed from the E programming language, Erlang, and Mozart/Oz.

Most of the rules described in the security section are easily implemented with non-shared, process-local message passing. The issues are more complex and require careful implementation design in the presence of shared state or cross-process communication. Shared memory will require efficient synchronisation primitives.

Messages carried between processes or platforms via a network protocol should use CAMP, the COLA Actor Messaging Protocol, which is yet to be defined. CAMP/UDP/IP would be a portable implementation that is friendly to Internet routers and packet filters. With careful design, cryptographic techniques can be used to implement flow control and privacy over the network.

7 References

[Piumarta09] Ian Piumarta. *COLA Kernel Abstraction*, VPRI Research Memo, 2009.