CS 420 Advanced Programming Languages
Fall Semester, 2022
Doc 18 Actor
Nov 1, 2022

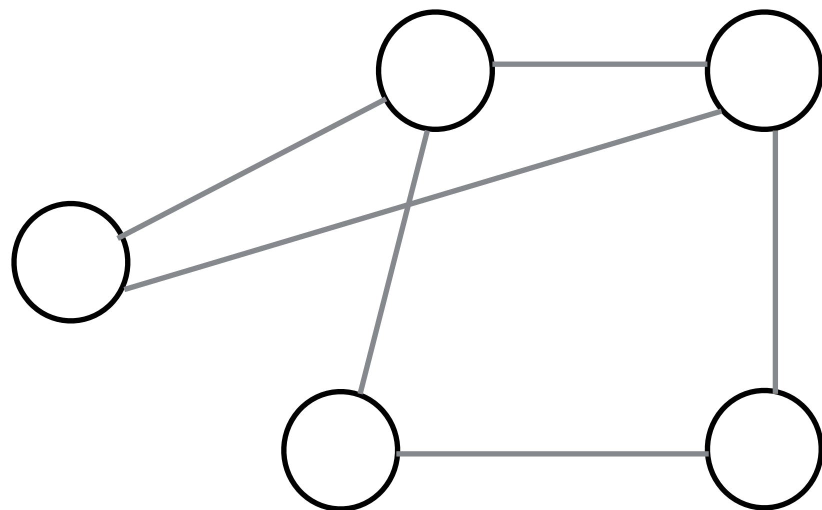
Copyright ©, All rights reserved. 2022 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/openpub/) license defines the copyright on this document.

Big Idea

Don't have one big program with a tangle of threads!

Have separate objects (actors) - mini-programs

- Each actor runs sequentially
- An actor receives "requests" on a queue or channel
- An actor can send results and requests to another actor via a queue or channel
- Each actor runs in a separate thread



Communicating Sequential Processes - CSP

1978 - C. A. R. Hoare first described

Mathematical theory of concurrency

- Message passing & Channels

Used to specify & verify Concurrent systems

- T9000 Transputer

Influenced design of programming languages

- Occam
- Go

Agents & Core.Async

Clojure implementations of the Actor idea

Agent

- Data
- Functions are sent to the data
- Data lives in is own thread, function evaluated in that thread

Core.Async

- Actor is code
- Each actor is in its own thread
- Data is sent to an actor

Channel

Communication link between producers and consumers

Channels can be
- Unbuffered
- Buffered

Types of Buffers

buffer
  blocks/parks when full

dropping-buffer
  While full drops items that are added

sliding-buffer
  While full drops oldest item when new item added

Producing a Channel

```
(chan)
(chan buf-or-n)
```

(chan 5)	channel with buffer of size 5
(chan (buffer 3))	channel with buffer of size 3
(chan (dropping-buffer 6))	
(chan (slidding-buffer 2))	

Reading/Writing Channels

`(>!! channel value)`

Writes value to channel

Blocks if buffer is full (unless buffer is sliding or drop)

`(<!! channel)`

Reads a value from channel

Blocks if nothing is available

Returns nil if channel is closed

Example

```
(def test-channel (async/chan 2))

(async/>!! test-channel "hello there")

(async/<!! test-channel)
```

Running in other Threads

futures
async/thread
go block

async/thread

(thread & body)

Runs body in separate thread

```
(async/thread (println "Hello"))
```

```
(def adder (async/thread (+ 1 2)))
(async/<!! adder)                        returns 3
```

```clojure
(defn producer
  [channel name]
  (doseq [x [1 2 "end"]]
    (do
      (Thread/sleep 100)
      (println  name "producing " x)
      (async/>!! channel x)))
  (async/close! channel))
```

```clojure
(let [channel (async/chan 7)]
  (println "Start")
  (async/thread (producer channel "a"))
  (async/thread (producer channel "b"))
  (async/thread (consumer channel))
  (println "consumer started"))
```

```clojure
(defn consumer
  [channel]
  (let [input (async/<!! channel)]
    (println "input" input)
    (when input
      (recur channel))))
```

```
Start
consumer started
=> nil
ba  producing  1producing  1

input 1
input 1
b producing  2
a producing  2
input 2
input 2
b a producing  end
producing  end
input end
input end
input nil
```

Rock Paper Scissors Example

```
(def MOVES [:rock :paper :scissors])
(def BEATS {:rock :scissors, :paper :rock, :scissors :paper})


(defn winner
  "Based on two moves, return the name of the winner."
  [[name1 move1] [name2 move2]]
  (cond
   (= move1 move2) "no one"
   (= move2 (BEATS move1)) name1
   :else name2))
```

Report - Helper

```
(defn report
  "Report results of a match to the console."
  [[name1 move1] [name2 move2] winner]
  (println)
  (println name1 "throws" move1)
  (println name2 "throws" move2)
  (println winner "wins!"))
```

Player

```
(defn rand-player
  "Create a named player and return a channel to report moves."
  [name]
  (let [out (async/chan)]
    (async/go (while true (async/>! out [name (rand-nth MOVES)])))
    out))
```

Judging results

```
(defn judge
  "Given two channels on which players report moves, create and return an
   output channel to report the results of each match as [move1 move2 winner]."
  [p1 p2]
  (let [out (async/chan)]
    (async/go
     (while true
       (let [m1 (async/<! p1)
             m2 (async/<! p2)]
         (async/>! out [m1 m2 (winner m1 m2)]))))
    out))
```

Playing single game

```
(defn init
  "Create 2 players (by default Alice and Bob) and return an output channel
of match results."
  ([] (init "Alice" "Bob"))
  ([n1 n2] (judge (rand-player n1) (rand-player n2))))

(defn play
  "Play by taking a match reporting channel and reporting the results of the latest
match."
  [out-chan]
  (apply report (async/<!! out-chan)))

  (play (init))
```

Playing Multiple Games

```
(defn play-many
  "Play n matches from out-chan and report a summary of the results."
  [out-chan n]
  (loop [remaining n
         results {}]
    (if (zero? remaining)
      results
      (let [[m1 m2 winner] (async/<!! out-chan)]
        (recur (dec remaining)
               (merge-with + results {winner 1}))))))
```

Multiple Games

```
(play-many game 10000)
```

```
{"Alice" 3323, "Bob" 3326, "no one" 3351}

"Elapsed time: 650.433 msecs"
```

rock paper scissors lizard spock

Try modifying code to play “rock paper scissors lizard spock”