COMMUNICATING SEQUENTIAL PROCESSES

Edward Cho
@zerokarmaleft

(

ABOUT ME

- Senior Software Engineer
  at Laureate Institute for Brain Research
- functional programmer since 2009
  (Clojure, Haskell, etc.)

COMMUNICATING SEQUENTIAL PROCESSES

- motivation
- API primitives
- examples

MOTIVATION

- decompose problems and express solutions as concurrent processes

- compose processes to solve bigger problems

- execute serially or in parallel

- revert inversion of control (no more callback hell!)

- “Good programs should be made out of processes and queues.” - Rich Hickey

CALLBACKS

```
var async = require(‘async’);

User.find(userId, function(err, user) {
  if (err) return errorHandler(err);
  User.all({where: {id: user.friends}}), function(err, friends) {
    if (err) return errorHandler(err);
    async.each(friends, function(friend, done) {
      friend.posts = [];
      Post.all({where: {userId: {$in: friend.id}}}, function(err, posts) {
        if (err) return errorHandler(err);
        async.each(posts, function(post, donePosts) {
          friend.push(post);
          Comments.all({where: post.id}, function(err, comments) {
            if (err) donePosts(err);
            post.comments = comments;
            donePosts();
          });
        }, function(err) {
          if (err) return errorHandler(err);
          done();
        });
      });
    }, function(err) {
      if (err) return errorHandler(err);
      render(user, friends);
    });
  }
});
```

PROMISES

```
var fs         = require('fs'),
    path       = require('path'),
    Q          = require('Q'),
    fs_readdir = Q.denodeify(fs.readdir),
    fs_stat    = Q.denodeify(fs.stat);

module.exports = function(dir) {
  return fs_readdir(dir)
    .then(function(files) {
      var promises = files.map(function(file) {
        return fs_stat(path.join(dir, file));
      });
      return Q.all(promises).then(function(stats) {
        return [files, stats];
      });
    })
    .then(function(data) {
      var files   = data[0];
      var stats   = data[1];
      var largest = stats
            .filter(function(stat) { return stat.isFile(); })
            .reduce(function(prev, next) {
              if (prev.size > next.size) {
                return prev;
              } else {
                return next;
              }
            });

      return files[stats.indexOf(largest)];
    });
};
```

CHANNELS

- like queues
- decouples consumers and producers
- multiple consumers, multiple producers
- reading/writing is a blocking operation
- unbuffered/buffered with fixed size

COMMUNICATING SEQUENTIAL PROCESSES

- C.A.R. Hoare 1978
- first-class channels
  - put/take operations
  - choice operation
  - timeout channels
- coordination via communication

JS-CSP

- ClojureScript core.async API is a strict subset of Clojure core.async API

- both are implemented using Clojure macros

- js-csp API is inspired by core.async

- implemented using channels and ES6 generators

CHANNELS

```
chan()    // create an unbuffered channel
chan(10)  // create a buffered channel with size 10

put(named-ch value)  // parking put
take(named-ch)       // parking take

close(named-ch)      // close a channel
```

GO

```
var c = chan();

go(function*() { yield put(c, ‘Hello!’); }
go(function*() {
  var msg = yield take(c);
  console.log(msg);
});
```

VENDING MACHINE

```
<div id=‘coin-slot’>
  <button onClick={this.handleClick}>Insert Coin</button>
</div>
<div id=‘candy-dispenser’>
  <button disabled={true}>Take Candy</button>
</div>
```

VENDING MACHINE

```
handleClick: function(e) {
  var coins = this.props.coins;
  go(function*() { yield put(coins, e); });
}
```

VENDING MACHINE

```
takeOneCoin: function*(coins, chocolates) {
  yield take(coins);
  this.setState({ coinInserted: true });
},

componentDidMount: function() {
  go(this.takeOneCoin, [this.props.coins,
                        this.props.chocolates]);
}
```

VENDING MACHINE

```
<div id=‘coin-slot’>
  <button disabled={this.state.disabled || this.state.coinInserted}
          onClick={this.onCoinInserted}>Insert Coin</button>
</div>
<div id=‘candy-dispenser’>
  <button disabled={this.state.disabled || this.state.coinInserted}
          onClick={this.onCandyTaken}>Take Candy</button>
</div>
```

VENDING MACHINE

```
onCoinInserted: function(e) {
  var coins = this.props.coins;
  go(function*() { yield put(coins, e); });
},

onCandyTaken: function(e) {
  var chocolates = this.props.chocolates;
  go(function*() {
    var candy = yield take(chocolates);
    console.log(‘Yay! Got a: ‘ + candy);
  });
}
```

VENDING MACHINE

```
serveTwoCustomers: function*(coins, chocolates) {
  for (var i = 0; i < 2; i++) {
    yield take(coins);
    this.setState({ coinInserted: true });
    yield put(chocolates, ‘chocolate’);
    this.setState({ coinInserted: false });
  }

  this.setState({ disabled: true });
}
```

VENDING MACHINE

```
serveCustomers: function*() {
  while (true) {
    yield take(coins);
    this.setState({ coinInserted: true });
    yield put(chocolates, ‘chocolate’);
    this.setState({ coinInserted: false });
  }
}
```

VENDING MACHINE

- one type of candy is boring
- the customer doesn’t have any choices
- extended alphabet
  { coin, chocolate, toffee }

CHOICE

```
// tries each channel operation in non-deterministic order
// returns an object with value and channel properties
var result = alts([take-ch, [put-ch ‘some value’]]);

if (result.channel === take-ch) {
  <do something useful…>
} else if (result.channel === put-ch) {
  <do something useful…>
}
```

VENDING MACHINE

```
serveCustomers: function*(coins, chocolates, toffees) {
  while (true) {
    yield take(coins);
    this.setState({ coinInserted: true });
    yield alts([[chocolates, ‘chocolate’],
                [toffees, ‘toffee]]);
    this.setState({ coinInserted: false });
  }
}
```

TIMEOUTS

```
timeout(n) // create a channel that closes after n milliseconds
```

TIMEOUTS

```
var ticks = chan();

go(function*() {
  while (true) {
    yield take(timeout(1000)); // parks for 1 second
    yield put(ticks ‘tick…’);  // send a message
  }
}
go(function*() {
  for (var i = 0; i < 10; i++) {
    var msg = yield take(ticks);
    console.log(msg);
  }
}
```

VENDING MACHINE

```
serveCustomers: function*(coins, chocolates, toffees) {
  while (true) {
    yield take(coins);
    this.setState({ coinsInserted: this.state.coinsInserted + 1 });
    var candy = yield alts([[chocolates, ‘chocolate’],
                            [toffees, ‘toffee’]]);
    if (candy.channel === chocolates) {
      this.setState({ chocolatesDispensed: this.state.chocolatesDispensed + 1 });
    } else if (candy.channel === toffees) {
      this.setState({ toffeesDispensed: this.state.toffeesDispensed + 1 });
    }
  }
}
```

VENDING MACHINE

```
customers: function*(names, coins, chocolates, toffees) {
  yield take(timeout(Math.random() * 5000));
  var candies = (Math.random() > 0.5) ? ‘chocolate’ : ‘toffee’;
  yield put(coins, 1);
  var candy = yield take(candies);
  console.log(name + ‘: got a ‘ + candy + ‘.’);
}
```

VENDING MACHINE

```
componentDidMount: function() {
  var coins      = this.props.coins;
  var chocolates = this.props.chocolates;
  var toffees    = this.props.toffees;

  for (var i = 0; i < this.props.size; i++) {
    go(this.customer, [‘[Customer ‘ + i + ‘]’,
                       coins, chocolates, toffees]);
  }
}
```

VENDING MACHINE

```
componentDidMount: function() {
  var coins      = this.props.coins;
  var chocolates = this.props.chocolates;
  var toffees    = this.props.toffees;

  for (var i = 0; i < this.props.gridSize; i++) {
    for (var j = 0; j < this.props.size; j++) {
      go(this.customer, [‘[Customer ‘ + (i * j) + ‘]’,
                         coins, chocolates, toffees]);
    }
  }
}
```

VENDING MACHINE

```
customers: function*(names, coins, chocolates, toffees) {
  yield take(timeout(Math.random() * 5000));
  var candies = (Math.random() > 0.5) ? ‘chocolate’ : ‘toffee’;
  yield put(coins, 1);
  yield take(timeout(Math.random() * 1000));
  var candy = yield take(candies);
  console.log(name + ‘: got a ‘ + candy + ‘.’);
}
```

BUFFERS

```
chan(n)          // creates a fixed buffer of size n
buffers.fixed(n) // also creates a fixed buffer of size n

buffers.dropping(n) // when full, new values put are dropped
buffers.sliding(n)  // when full, puts drop the oldest value
```

VENDING MACHINE

```
function makeChannels(n) {
  var channels = [];

  for (var i = 0; i < n; i++) {
    channels.push(chan(100));
  }

  return channels;
}
```

AND BEYOND

- load balancing
- replication
- consensus

AND BEYOND

- cancellation
  - resource clean-up
- catastrophic events
  - restart
  - checkpoints

AND BEYOND

- pipes
  - straight pipes
  - fan-in, fan-out
  - pub-sub
  - dynamic taps

AND BEYOND

- transducers
  - composable logic
  - combined with CSP machinery
  - easily testable

GOTCHAS

- component's state machine should be encapsulated in a goroutines
- all synthetic events should simply feed channels
- send immutable data on channels
- create channels at lowest "supervisor" component possible, pass to child components via properties
- as always, beware scope issues with `this`

REFERENCES

- Tony Hoare, Communicating Sequential Processes
- Nguyễn Tuấn Anh, https://github.com/ubolonton/js-csp
- Rich Hickey,“The Language of the System”
  Clojure conj 2012
- Rich Hickey,“Clojure core.async”
  Strange Loop 2013
- Rob Pike,“Concurrency is not Parallelism”
- Tim Baldridge,“core.async”
  Clojure conj 2013
- James Long,“Taming the Asynchronous Beast with CSP in JavaScript”
  http://jlongster.com
- David Nolen,“ES6 Generators Deliver Go Style Concurrency”
  http://swannodette.github.io

)