Synchronous vs. Asynchronous Programming

Jan Pascal Maas

Institute for Software Engineering and Programming Languages
University of Luebeck

25. January 2016

Seminar Concepts of Programming Languages

Agenda

Introduction

- programs rely on multiple constraints
  - especially *concurrency* and *communication* increase complexity
- concurrency is largely explored
  - general programming paradigms can be used
  - multiple approaches exist

Introduction—Communication

- increases complexity largely

Example (Air Traffic Control System)

- information is known beforehand
  - all tasks can run independently
  - low complexity

Introduction—Communication

- increases complexity largely

Example (Air Traffic Control System)

- information is known beforehand
  - all tasks can run independently
  - low complexity
- information and communication during execution
  - system needs to be capable of accepting and processing information
  - requires high amount of synchronized communication

Introduction—Paradigms to Synchronize

- *blocking* (scheduler-based): block task to use resources differently
  - blocked task ensures resuming of computation
  - specific synchronization condition required
- *busy-waiting*: use of an evaluation loop
  - reevaluated specific condition till it becomes true

Synchronous Programming

- applies scheduler-based synchronization
  - blocks a task if it is necessary
  - resources can be used for a different task
  - if needed resources are available, computation continues
- ensures correctness of the system

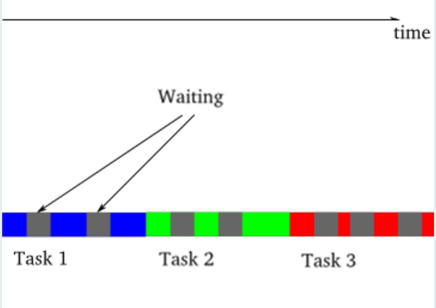

Figure: Synchronous blocking model [1].

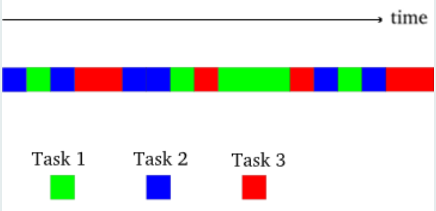

Figure: Asynchronous model [1].

Asynchronous Programming

- uses busy-waiting
  - re-evaluation of a specific condition
  - if true, condition depends on an external system
  - actions to compute are specified *before* execution
- main thread continues running
- actions that depend on external system(s) are executed on different thread

Synchronous Approaches—*LUSTRE*

- data-flow oriented language
- focussed on temporal correctness
- variables are treated as infinite sequences

$$(x_0 = e_0,\ x_1 = e_1,\ \ldots,\ x_n = e_n, \ldots)$$

LUSTRE—Operators

Let $X = (x_0, x_1, \ldots, x_n, \ldots)$ and $Y = (y_0, y_1, \ldots, y_n, \ldots)$.

- pre(X) $= (\text{nil}, x_0, x_1, \ldots, x_{n-1}, \ldots)$
- X -> Y $= (x_0, y_1, \ldots, y_n, \ldots)$

LUSTRE—Operators

Let $X = (x_0, x_1, \ldots, x_n, \ldots)$ and $Y = (y_0, y_1, \ldots, y_n, \ldots)$.

- pre(X) = (nil, x_0, x_1, …, x_{n-1}, …)
- X -> Y = (x_0, y_1, …, y_n, …)

E	=(	e_0	e_1	e_2	e_3	e_4	e_5	…	)
B	=(	*tt*	*ff*	*tt*	*tt*	*ff*	*ff*	…	)
X = E when B	=(	$x_0 = e_0$		$x_1 = e_2$	$x_2 = e_3$			…	)
Y = current(X)	=(	e_0	e_0	e_2	e_3	e_3	e_3	…	)

LUSTRE—Synchronization

- when is used to create *streams*
- streams allow synchronization of the program
- to synchronize differently clocked streams, current is used

LUSTRE—Synchronization

- when is used to create *streams*
- streams allow synchronization of the program
- to synchronize differently clocked streams, current is used
- *assertions* generalize equations → facts to synchronize program

```
assert not (x and y)
```

SIGNAL

- concept similar to LUSTRE
- allows explicit synchronization using synchro
- merging of two signals with default

ESTEREL

- imperative language
- variables are called *signals*
- *reaction*: process of computing output based on input
- fixed status and current value (initially $\perp$) in the same reaction

ESTEREL—Synchronization

- emit: sending output signals
- watching: await specific signal
- present: detects for presence of a signal

ESTEREL—Synchronization

- emit: sending output signals
- watching: await specific signal
- present: detects for presence of a signal

```
do
    I1 -> O1
watching I2;
emit O2
```

Asynchrony through Concurrency

- multiple threads used to perform IO-bound tasks

Asynchrony through Concurrency

- multiple threads used to perform IO-bound tasks
- Problem: Scalability is limited
- requirement of different approaches

Asynchronous Approaches—Event Loop

- inversion of control
- efficiency and scalability
- control over switching between application activities
- relies on notification facilities
- application handles occurrence of events
- commonly used: *Node.js*

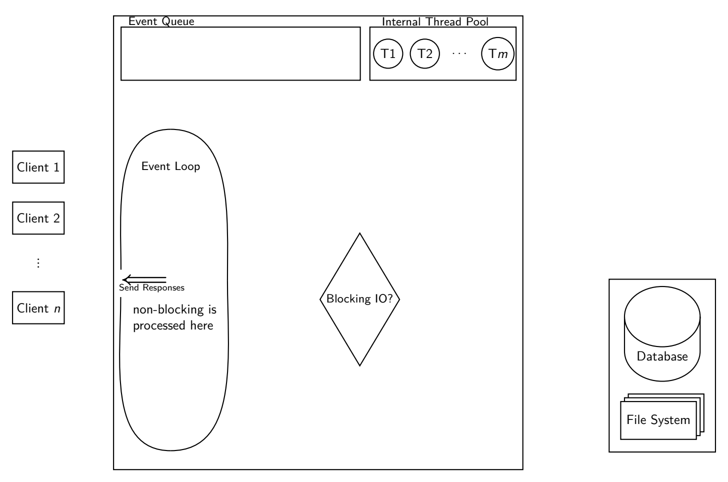

Figure: Node.js processing model [2].

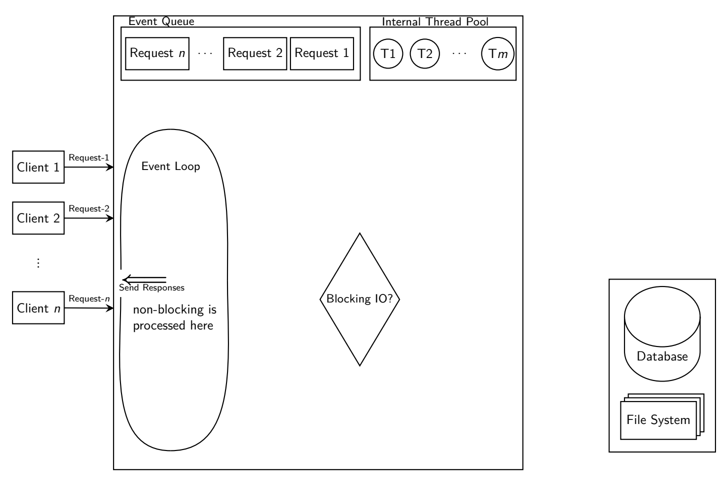

Figure: Node.js processing model [2].

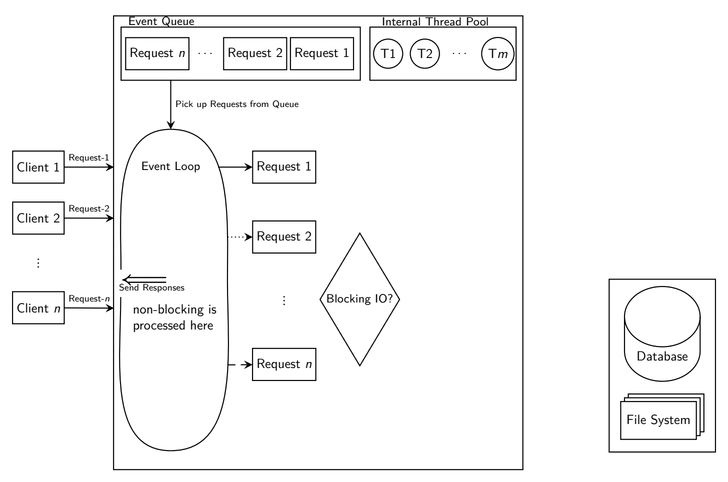

Figure: Node.js processing model [2].

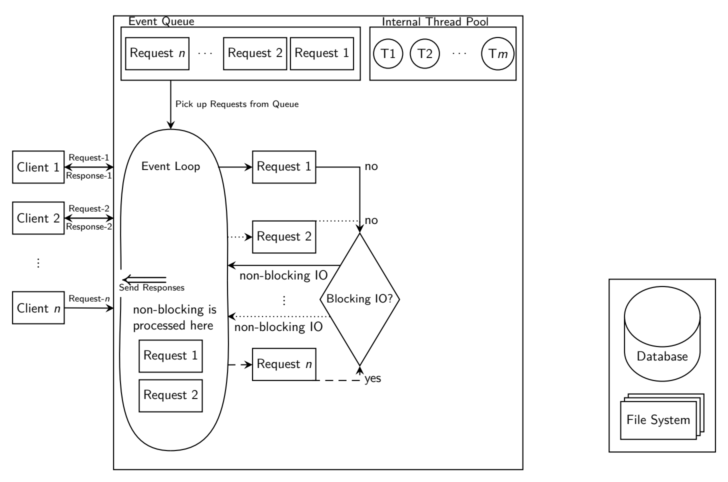

Figure: Node.js processing model [2].

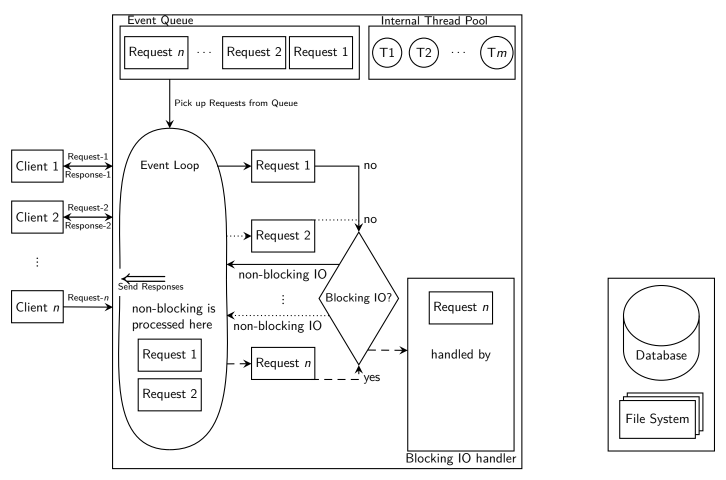

Figure: Node.js processing model [2].

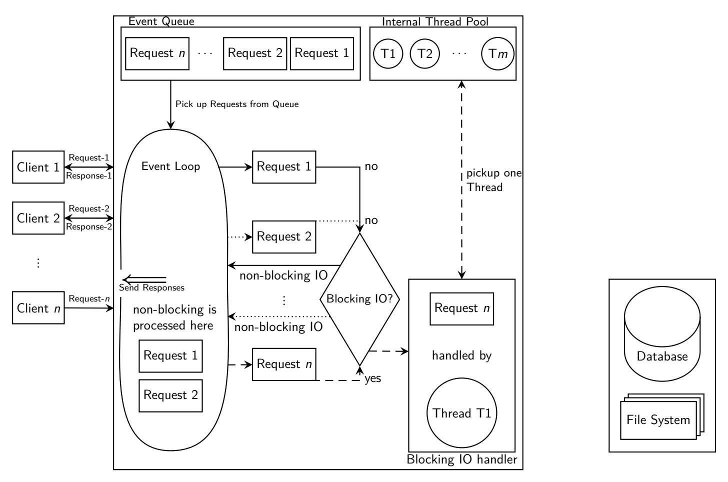

Figure: Node.js processing model [2].

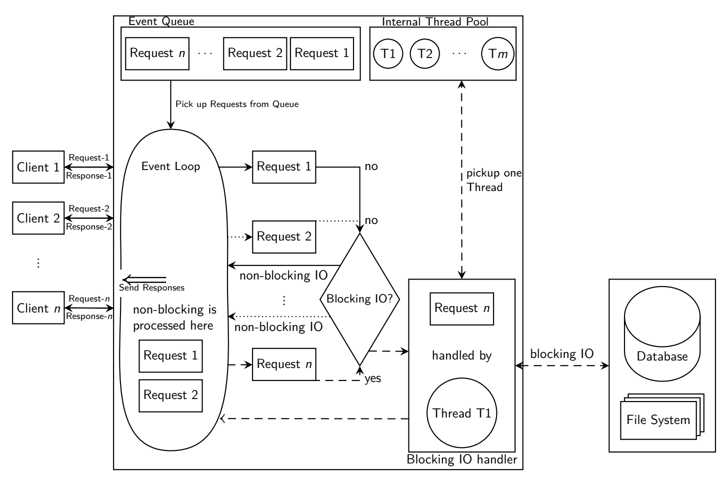

Figure: Node.js processing model [2].

Continuations

Example

```
public int Divide(int top, int bottom)
{
    if (bottom==0)
    {
        throw new InvalidOperationException("div␣by␣0");
    }
    else
    {
        return top/bottom;
    }
}

public bool IsEven(int aNumber)
{
    var isEven = (aNumber % 2 == 0);
    return isEven;
}
```

Continuations

Example

```
public T Divide<T>(int top, int bottom, Func<T> ifZero, Func<int,T> ifSuccess)
{
    if (bottom==0)
    {
        return ifZero();
    }
    else
    {
        return ifSuccess( top/bottom );
    }
}

public T IsEven<T>(int aNumber, Func<int,T> ifOdd, Func<int,T> ifEven)
{
    if (aNumber % 2 == 0)
    {
        return ifEven(aNumber);
    }
    else
    {
        return ifOdd(aNumber);
    }
}
```

Asynchrony in F#

- also implementing event-based paradigm
- library with new syntactic category *aexpr*
- use capability of language to handle different context
- asynchronous operations are capable of binding core language results

Asynchrony in F#—Task Generators

- sometimes functions need task generators
- can run asynchronous computations synchronously
- can be run as a co-routine if it does not produce a result

Example (Task Generators)

```
let sleepThenReturnResult =
    async { printfn "before␣sleep"
    do! Async.Sleep 5000
    return 1000
}

let res = Async.RunSynchronously sleepThenReturnResult
printfn "result␣=␣%d" res
```

Where is the callback run?

- in .NET any computation has access to *synchronization context*
- any callback is running "somewhere"

Where is the callback run?

- in .NET any computation has access to *synchronization context*
- any callback is running “somewhere”
- can be abused to run callbacks based on function closures

Asynchronous Resource Clean-Up

- language feature: use!
- allows to directly dispose resources
- cancellation of operations: implicit propagation of a token

Synchronous Programming—Pro and Contra

- ensures temporal and logical correctness

Synchronous Programming—Pro and Contra

- ensures temporal and logical correctness

- blocking a thread might block complete system
- requires manual use of synchronization mechanisms
- may lie outside the control of the programmer

Asynchronous Programming—Pro and Contra

- system stays responsive
- outperforms synchronous systems
- main process is *always* single-threaded
- programmer is in control of task suspension
- multi-threading is not forbidden
- allows to reduce the syntax to “computation expressions”
- no need to explicitly ensure *temporal correctness*

Asynchronous Programming—Pro and Contra

- system stays responsive
- outperforms synchronous systems
- main process is *always* single-threaded
- programmer is in control of task suspension
- multi-threading is not forbidden
- allows to reduce the syntax to "computation expressions"
- no need to explicitly ensure *temporal correctness*

- program needs to be organized in smaller steps
- no explicit use of multi-threading
- not all interprocess communication can be reduced to events
- high complexity without callbacks

Conclusion

- Asynchronous programming eliminates some issues from synchronous programming
- Asynchrony allows the programmer to take control
- Asynchrony takes care of temporal correctness

Conclusion

- Asynchronous programming eliminates some issues from synchronous programming
- Asynchrony allows the programmer to take control
- Asynchrony takes care of temporal correctness

- Synchronous programming is required in some areas
- Asynchronous programming can not be used at any operation
- Complexity of Asynchrony can ensure unmaintainable code

Thank You for Your attention.

Literature

Dave Peticolas.
An introduction to asynchronous programming and twisted.
http://krondo.com/?p=1209, 2009.
Accessed: 2015-11-07.

Rambabu Posa.
Node.js processing model – single threaded model with event loop architecture.
https://tinyurl.com/jjc7btk, 2015.
Accessed: 2015-12-17.