Multi-Agent Simulation

Autonomous Software Agents - Lab

Marco Robol - marco.robol@unitn.it

Contents

- **Observable** - Sensors will be implemented on top of Observable events/facts - Agents can subscribe to given facts and being notified when changes happen.
- **Simulation loop** - Must be non-blocking. Implemented on top of the node.js event loop. For example, consider using setTimeout().
- **Intention/Plan execution**: Non-blocking implementation. Code splitted as much as possible so to keep the agent loop spinning.
- **Modularity**: agent as a set of associated Intention<->Goal. Given a goal, the agent search for an applicable intention in his own plan set. Different intentions could be used to achieve the same desire.

Repository: https://github.com/marcorobol/Autonode.js

Observable #1

An example:

```
var myHouse = new Observable( { mainLightOn: true, carCharging: true } )

// get value
myHouse['mainLightOn'] // true

let observer = value => console.log('mainLightOn', value)
o1.observe('mainLightOn', observer)

// set value
o1.mainLightOn = false
// observer callback: mainLightOn false
```

> ./src/utils/Observable.js

Observable #2

```
class Observable {
    constructor (init) {...}
    set (key, value) {...
        this.#map[key].value = value;
        // Postpone observer callbacks, queue as microtask!
        Promise.resolve().then( () => {
            for (let o in this.#map[key].observers)
                this.#map[key].observers[o](v, key);
        }).catch( err => console.error(err) )
    ...}
    observe (key, observer) {...}
    unobserve (key, observer) {...}
    async notifyChange (key) {...}
}
```

Observable #3

Additional examples:

```
// ...
o1.unobserve('mainLightOn', observer)
o1.set('person_in_room', 'kitchen')
```

```
// promises
notifyChange('person_in_room')
    .then( (value) => console.log('person moved into', value))
```

```
// async/await syntax
let value = await notifyChange('person_in_room')
console.log('person moved into', value)
```

Clock

```
var clock = new Observable( {dd: 0, hh: 0, mm: 0} )
while(true) {
    // Postpone loop: queue as macrotask!
    await new Promise( res => setTimeout(res, 50))
    if(clock.mm<60-15)
        clock.mm += 15;
    else {
        if(clock.<24) {
            clock.hh += 1; // increased hh but mm still 45
            clock.mm = 0; // at the time observers on hh are called also mm are updated
        }
        else {
            clock.mm = 0; clock.hh = 0; clock.dd += 1
        }
    }
}
```

> ./src/utils/Clock.js

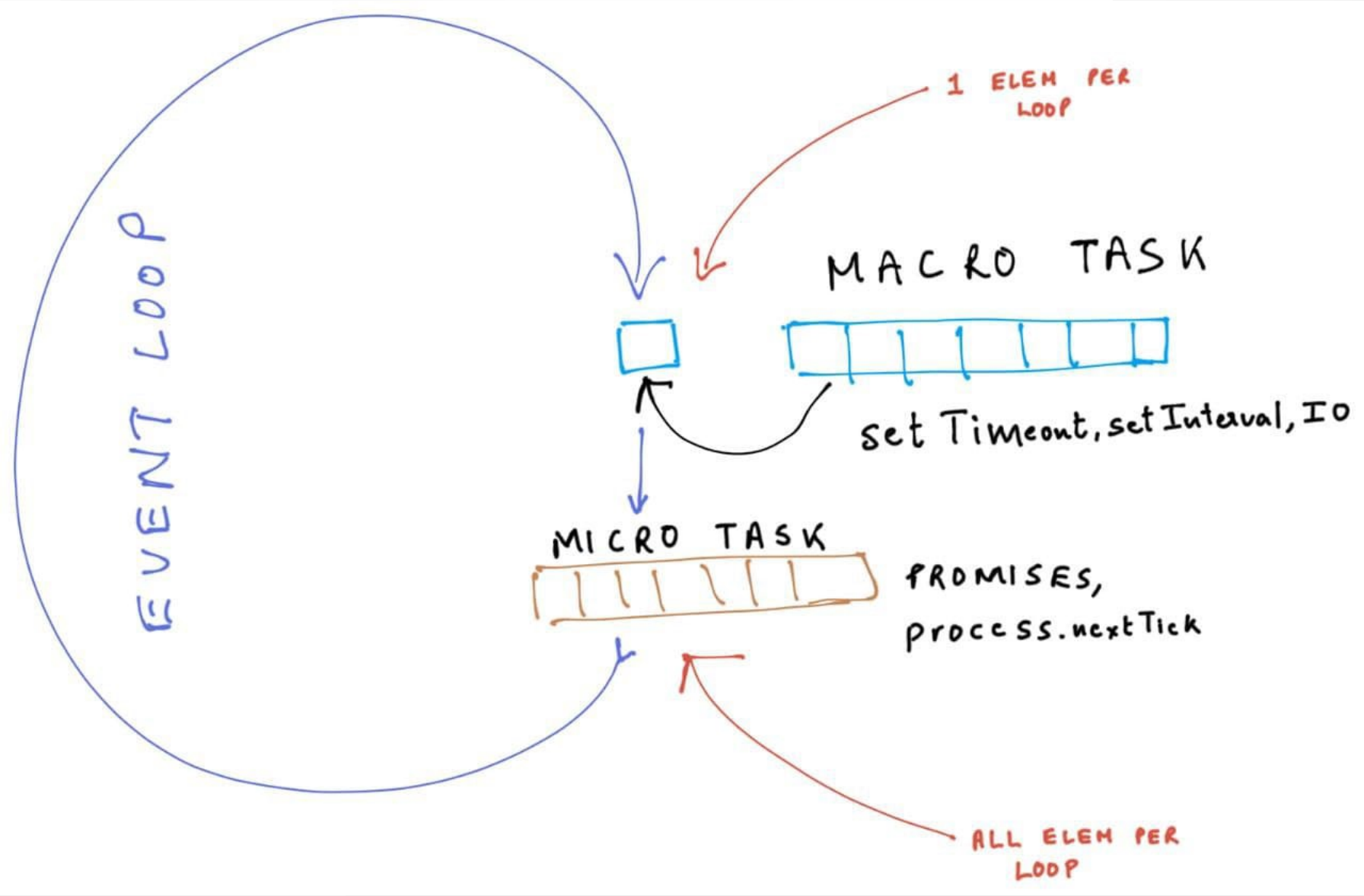
Macrotasks and Microtasks

If microtasks continuously add more elements to microTasks queue, macroTasks will stall and won't complete event loop in shorter time causing event loop delays.

Check this: https://medium.com/dkatalis/eventloop-in-nodejs-macrotasks-and-microtasks-164417e619b9.
Similarly it is for javascript on browser-side: https://medium.com/@idineshgarg/let-us-consider-an-example-a58bb1c11f55

```
// microtask, this would still postpone all timers and IO from being executed!!!
await Promise.resolve();
```

```
// macrotask, queues with other timers and IO
await new Promise( res => setTimeout(res, 0))
```


EVENT LOOP
1 ELEM PER LOOP
MACRO TASK
set Timeout, setInterval, IO
MICRO TASK
PROMISES,
process.nextTick
ALL ELEM PER LOOP

Listen to clock events

```
// Daily schedule
Clock.global.observe('mm', (key, mm) => {
    var time = Clock.global
    if(time.hh==12 && time.mm==0)
        house.people.bob.in_room = 'kitchen'
    if(time.hh==13 && time.mm==30)
        house.people.bob.in_room = 'living_room'
    if(time.hh==19 && time.mm==0)
        house.people.bob.in_room = 'kitchen'
    if(time.hh==20 && time.mm==15)
        house.people.bob.in_room = 'living_room'
})
```

Goal

```
class Goal {

    constructor (parameters = {}) {...}

}
```

> ./src/bdi/Goal.js

Intention *exec

```
class DimOnLight extends Intention {
    *exec () {
        let l = this.goal.parameters.l
        for (let i = 0; i <= 10; i++) {
            this.log(l, i);
            house['lightOn '+l] = i/10;
            yield new Promise( res => setTimeout(res, 50));
        }
        house['lightOn '+l] = 1
    }
}
await new DimOnLight(new Goal({l: 'light1'})).run()
```

> ./src/bdi/Intention.js

Intention yield

```
var s1, s2, s3, s4;
class DimOnLight extends Intention {
    *exec () {
        let s1 = yield new Promise( res => setTimeout(res, 50))  // promise
        let s2 = yield 1234                                      // value
        let s3 = yield false                                     // false
        let s4 = yield                                           // none
    }
}
await new DimOnLight(new Goal()).run()
// s1 = undefined
// s2 = 1234
// s3 = false
// s4 = undefined
```

Intention execution: run()

```
async run () {
    var iterator = this.exec(); var yieldValue = null; var failed = false; var done = false;
    while (!failed && !done) {

        // passing a value or waiting for the promise to resolve into a value
        var {value: yieldValue, done: done} = iterator.next(await yieldValue)

        // attach immediately a catch callback to avoid getting a PromiseRejectionHandledWarning
        if (yieldValue instanceof Promise)
            yieldValue.catch( err => { console.error(err.stack || err); failed = true; return false; } );

        // Always wait for a timer to avoid stopping the event loop within microtask queue!
        await new Promise( res => setTimeout(res, 0))
    }
    if (done && !failed)
        return true;
    else
        return false; // Since we are in an aync function, here we are rejecting the promise. We will need to catch this!
    }
}
```

> ./src/bdi/Agent.js

Agent

Same goal -> possible different implementation

```
async postSubGoal (subGoal) {
    for (let intentionClass of Object.values(this.intentions)) {
        if (!intentionClass.applicable(subGoal))
            continue; // if not applicable try next intention

        this.log('Trying to use intention', intentionClass.name, 'to achieve goal', subGoal.toString())
        var intention = new intentionClass(this, subGoal)

        var success = await intention.run().catch( err => {this.log('Error in run() intention:', err)} )
        if ( success ) {
            this.log('Succesfully used intention', intentionClass.name, 'to achieve goal', subGoal.toString())
            return Promise.resolve(true) // same as: return true;
        }
        else {
            this.log('Failed to use intention', intentionClass.name, 'to achieve goal', subGoal.toString())
            continue; // retrying
        }
    }
    this.log('No success in achieving goal', subGoal.toString())
    return Promise.resolve(false) // different from: return false; which would reject the promise!!!
}
```

Example

scenario3.js

> ./src/houseworld/scenario3.js

Assignment 2

Assignment 2 consists in an initial implementation of your system. It can build on top of the code discussed during the lecture (https://github.com/marcorobol/Autonode.js). Provide a README.md file introducing your code (it is nice to have pointers to files you implemented or modified) and explaining how to run a short demo of the system (some messages logged on the console are enough).

Submission form: https://forms.gle/9TMbekGPBSboHJRy8.

Attach a .zip file with your source code. Please exclude *node_modules* folder.

Submission deadline is the 3/05/2022 at midnight!

Next

- **Planning** - Fact-based representation of the environment should include house structure and devices (status and actions). For example: door(kitchen, living_room), stairs(kitchen, garage), light_on(bedroom), car_charging(car1)
- **Actions and Sensors** actions effects applied by the environment (cannot be forced by the agent). Sensors define events to be notified to agents.

Thank you

Questions?

marco.robol@unitn.it

Repository: https://github.com/marcorobol/Autonode.js