MINISTERUL EDUCAȚIEI ŞI CERCETĂRII ŞTIINȚIFICE

UNIVERSITATEA „1 DECEMBRIE 1918” DIN ALBA IULIA

FACULTATEA DE ŞTIINTE EXACTE ŞI INGINEREŞTI

MASTER: PROGRAMARE AVANSATĂ ŞI BAZE DE DATE

CONCURRENCY IN OBJECT ORIENTED VERSUS FUNCTIONAL PROGRAMMING LANGUAGES

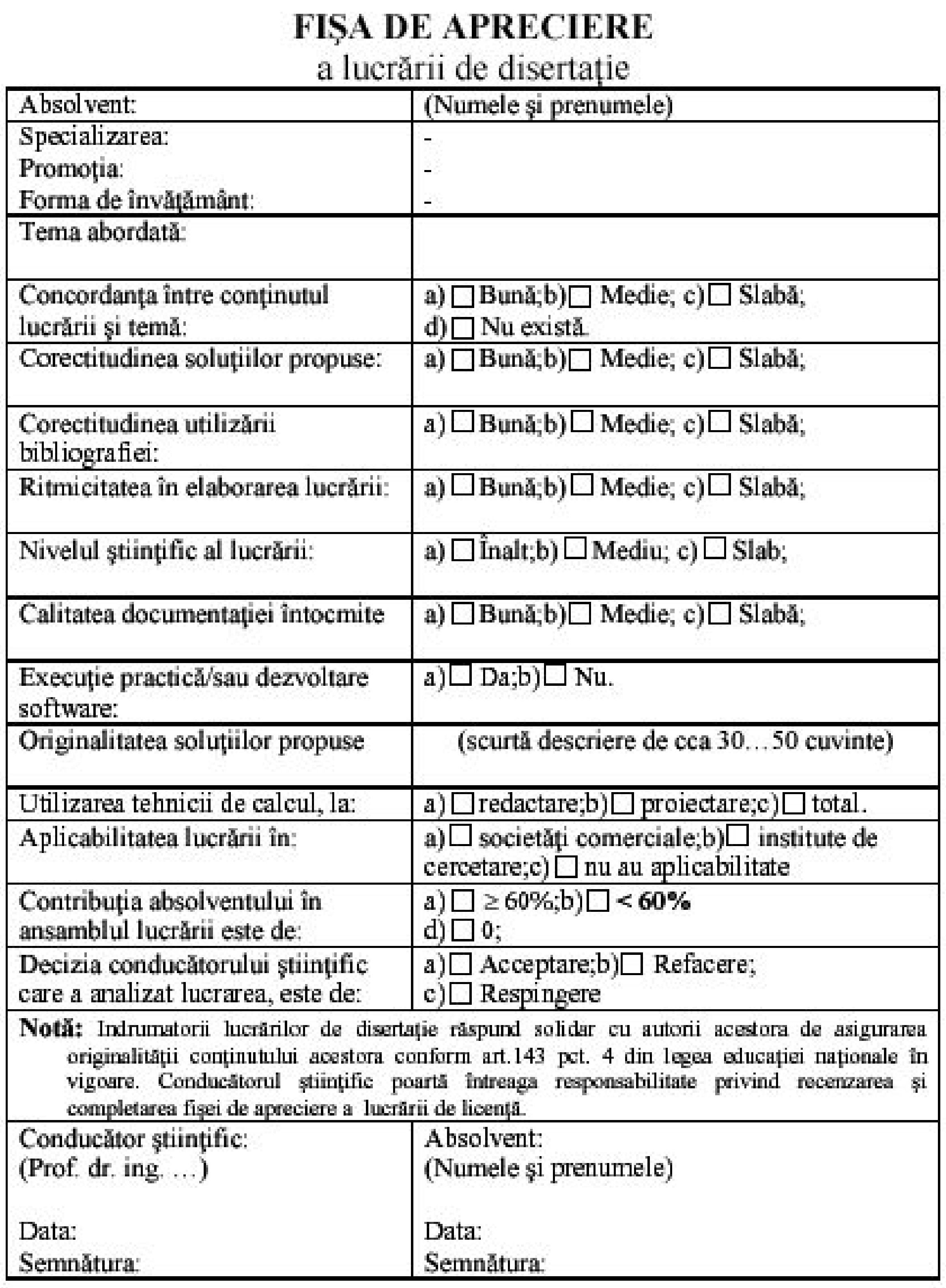
COORDONATOR ŞTIINȚIFIC,
CONF. UNIV. DR. CORINA ROTAR

ABSOLVENT,
student VASILE ADRIAN ROȘIAN

ALBA IULIA

2015

Adeverinţă alegerea temei



FIŞA DE APRECIERE
a lucrării de disertaţie

Absolvent:	(Numele şi prenumele)
Specializarea: Promoţia: Forma de învăţământ:	- - -
Tema abordată:	
Concordanţa între conţinutul lucrării şi temă:	a) ☐ Bună;b) ☐ Medie; c) ☐ Slabă; d) ☐ Nu există.
Corectitudinea soluţiilor propuse:	a) ☐ Bună;b) ☐ Medie; c) ☐ Slabă;
Corectitudinea utilizării bibliografiei:	a) ☐ Bună;b) ☐ Medie; c) ☐ Slabă;
Ritmicitatea în elaborarea lucrării:	a) ☐ Bună;b) ☐ Medie; c) ☐ Slabă;
Nivelul ştiinţific al lucrării:	a) ☐ Înalt;b) ☐ Mediu; c) ☐ Slab;
Calitatea documentaţiei întocmite	a) ☐ Bună;b) ☐ Medie; c) ☐ Slabă;
Execuţie practică/sau dezvoltare software:	a) ☐ Da;b) ☐ Nu.
Originalitatea soluţiilor propuse	(scurtă descriere de cca 30…50 cuvinte)
Utilizarea tehnicii de calcul, la:	a) ☐ redactare;b) ☐ proiectare;c) ☐ total.
Aplicabilitatea lucrării în:	a) ☐ societăţi comerciale;b) ☐ institute de cercetare;c) ☐ nu au aplicabilitate
Contribuţia absolventului în ansamblul lucrării este de:	a) ☐ $\geq$ 60%;b) ☐ **< 60%** d) ☐ 0;
Decizia conducătorului ştiinţific care a analizat lucrarea, este de:	a) ☐ Acceptare;b) ☐ Refacere; c) ☐ Respingere
Notă: Indrumatorii lucrărilor de disertaţie răspund solidar cu autorii acestora de asigurarea originalităţii conţinutului acestora conform art.143 pct. 4 din legea educaţiei naţionale în vigoare. Conducătorul ştiinţific poartă întreaga responsabilitate privind recenzarea şi completarea fişei de apreciere a lucrării de licenţă.	
Conducător ştiinţific: (Prof. dr. ing. …) Data: Semnătura:	Absolvent: (Numele şi prenumele) Data: Semnătura:

DECLARAȚIE DE ONESTITATE

Subsemnatul (a) ....,

fiul lui .... şi al ...., identificat cu BI / CI seria .... nr.............., eliberat de ...., la data de ....,

CNP ...., declar pe proprie răspundere că la conceperea lucrării de disertație cu titlul ....

.. ....

.. ....

sub coordonarea ştiințifică a ....

nu am folosit alte surse decât cele menționate în bibliografie, lucrarea îmi aparține în întregime şi a fost redactată cu respectarea strictă a regulilor de evitare a plagiatului.

Alba Iulia,

…………….

Masterand,

Roşian Vasile Adrian

Table of contents

Introducere (Romanian)

Necesitatea demersului

Prezenta lucrare doreşte să fie un efort de popularizare şi de comparare a tehnicilor de structurare a programelor de calculator din prisma consurenţei.

Necesitatea acestui demers şi importanţa acestuia derivă din situaţia curentă la nivelul limbajelor de programare. Limbajele de programare curente se orientează tot mai mult pe productivitate şi ignoră parţial constrângerile legate de puterea limitată de procesare doar pentru a scoate cât mai repede pe piaţă caracteristici care să îi ajute pe programatori să producă software cu viteză mai mare. Din păcate acest trend nu este sustenabil întrucât puterea de procesare nu mai creşte conform legii lui Moore (datorită limitărilor fizice ale circuitelor electrice din procesoare) şi singura modalitate de a susţine nevoia tot mai mare de putere de procesare este prin paralelizarea operaţiunilor.

Aceasta paralelizare a operaţiunilor este însă dificil de realizat în limbaje de programare care nu au structurile necesare pentru a gestiona dificultăţile unei astfel de abordări, dar dificultatea constă şi în abordarea problemei din perspectiva unui mod de gandire impropriu.

Problematica discutată

Abordăm problema structurării programelor în vederea concurenţei operaţiunilor într-o manieră comparativă: punem faţă în faţă programarea imperativă, cu vârful său de lance - Programarea Orientată Obiect şi programarea funcţională. De notat faptul ca distingem între concurenţă şi paralelism - în timp ce prima este modalitatea de structurare a unui program astfel incât operaţiunile specifice programului să se desfăşoare în paralel din punct de vedere logic, a doua este efectiv modalitatea prin care la nivel hardware se realizează distribuirea sarcinilor pe unităţi de procesare. Întrucât interesul nostru este centrat pe software, ne interesează modalităţile de a implementa concurenţa în programele de calculator.

Începem abordarea problematicii prin prezentarea concurenţei în limbajele orientate obiect, cu exemplificarea printr-un program Python care foloseşte modulul de multi-processing al limbajului. Capitolul încearcă şi o prezentare non-exhaustivă a tehnicilor folosite atât pentru separarea logică a firelor de execuţie ale programului (înţelegându-se aici procesele şi *thread*urile) cât şi pentru protejarea memoriei folosite pentru comunicarea între aceste fire de execuţie. Sunt arătate aici şi capcanele comune legate de aceste mecanisme de siguranţă, cum sunt blocajul de sincronizare (*livelock*) şi blocajul de eliberare (*deadlock*) ca forme ale problemei mai generale cunoscute sub numele de deprivare de resurse (*resource starvation*). În acest context introducem şi conceptul cunoscut în literatura de specialitate sub numele de atomicitate (*atomicity*) cu referire la operaţiunile de update înţelese ca un ciclu de citire-scriere.

În capitolul următor ne aplecăm asupra surselor de dificultate în programarea aplicaţiilor ce folosesc concurenţa. Arătăm astfel că sursa principală a dificultăţilor de programare folosind abordarea concurentă este starea şi împrărţim starea (*state* in limba engleză) în stare implicită (generată de modelul de business) şi stare accidentală, conform literaturii citate. Încercăm să demonstrăm că limbajele de programare imperative (inclusiv cele orientate obiect) introduc o varietate specială de stare, generată de controlul ordinii de execuţie şi că acest lucru nu este necesar şi este generator de complexitate.

Tot în acest capitol arătăm şi că starea, atât cea implicită cât şi cea accidentală, este consecinţă a introducerii neobservate a factorului timp în program, ca un parametru care nu este tratat. Acest lucru este dăunător întrucât timpul este generatorul stării iar starea este sursa complexităţii (înţelegem prin stare valoarea momentană, o observaţie dintr-un şir continuu de posibile valori, fie că sunt valori citite dintr-un fişier sau date discrete introduse de un utilizator de la tastatură - a se vedea conceptul de *stream*).

Mergând mai departe cu raţionamentul postulăm că timpul este la rândul lui un construct uman, derivat din procesul de interacţiune a bio-procesorilor umani - sau interpretori, cum sunt cunoscuţi în paradigma procesual-organică - cu realitatea obiectivă care alterează capacitatea de procesare permiţând astfel perceperea informaţiilor suplimentare (şi care creeaza astfel iluzia cauza-efect aflată la baza percepţiei timpului). Timpul devine în acest context o formă de serializare a informaţiei pentru procesare sau o expresie a dependenţelor structurale ale elementelor realităţii. Introducerea sa în operaţiile programului (direct sau indirect) este cel puţin inoportună şi generatoare de complexitate.

În ultimul capitol tratăm limbajele funcţionale, ca alternativă superioară la limbajele imperative din perspectiva concurenţei. Limbajele funcţionale îşi au rădăcinile în conceptul matematic de funcţie şi sunt, prin urmare, declarative - un program scris într-un limbaj funcţional descrie transformările necesare datelor de intrare pentru a deveni date de ieşire, spre deosebire de limbajele imperative, unde sunt implementaţi paşii pentru ca datele să ajungă la forma dorită. Diferenţa este subtilă şi dă naştere unui concept legat de necesitatea unei procesări: într-un limbaj imperativ, procesarea are loc la fiecare pas şi din rezultatul procesării se extrag informaţiile dorite la final. În limbajele funcţionale abia când este necesar un rezultat se aplică transformările descrise asupra datelor pentru a obţine acest rezultat - *lazy evaluation*. Având în vedere această proprietate, putem construi o listă infinită de rezultate dorite, în care primul element este rezultatul nul şi elementele subsecvente sunt descrise doar de legitatea de transformare. Tot în acest fel se pot defini şi datele de intrare - primul element este lista goală şi elementul subsecvent e definit de legea de cerere a datelor de către rezultatul dorit, ce determină o interacţiune cu lumea de la marginea programului pentru a obţine următoarea valoare de intrare (ca expresie a dependeţei structurale a informaţiei de informaţia din realitatea exterioară programului). Aceste două liste poartă numele de *stream*-uri şi nu exprimă altceva decât faptul că timpul nu este necesar în procesare, el nefiind altceva decât serializarea realităţii de la marginea programului în legitatea de transformare a informaţiei (a se vedea conceptul matematic de domeniu şi co-domeniu, cu funcţia drept aplicaţia de transformare între ele).

Prezentăm apoi împărţirea limbajelor funcţionale în limbaje pure şi impure din punctul de vedere al tratării stării.

În limbajele impure starea nu este încă un parametru fucnţional şi poate fi găsită în corpul unei funcţii făra să fie prezentă în parametri, permiţând astfel o mai uşoară gestiune a efectelor secundare (logging, input-output, valori intermediare) şi un raţionament mai uşor pentru un programator venit din lumea programelor imperative. Totuşi, chiar şi în aceste limbaje impure, arătăm noi, există o schimbare de emfază, aceste modificări fiind strict controlate şi accentuate, pentru a sublinia faptul că sunt potenţiale surse de erori.

În limbajele funcţionale pure starea este parametru în semnătura funcţiei prin definirea unor tipuri de date speciale numite monade (efectiv interfeţe pentru interacţiunea cu starea prin funcţiile de *bind* şi *return*), tipuri de date care se aplică parametrilor de intrare pentru a returna igienic atât rezultatul dorit cât si modificarea realităţii generată de evaluarea funcţiei (spre exemplu input-ul este o modificare a realităţii - *acţiunea* de citire) de la marginea programului generată de necesitatea de obţinere a unei valori de intrare pentru obţinerea rezultatului.

Pentru a putea face în subcapitolul următor introducerea elementelor de concurenţă în limbajele funcţionale efectuăm de asemenea o scurtă trecere în revistă a modalităţilor de realizare a abstractizării şi compoziţiei în limbajele funcţionale. Inventariem astfel structurile de date ca modalitate de abstractizare şi compoziţia funcţională atât ca modalitate de abstractizare a funcţionalităţii, cât şi în calitate de element de compoziţie a programelor.

În sfârşit începem şi prezentarea concurenţei în limbajele de programare functionale. Întrucât însă dificultatea în structurarea concurentă a programelor nu constă în realizarea concurenţei, elementele de concurenţă sunt doar amintite în aceste ultime subcapitole, mai ales că acestea sunt foarte similare cu cele din limbajele imperative. Amintim astfel threadurile şi procesele, atât cele native (specifice sistemului de operare), cât şi cele virtuale (gestionate de către maşina virtuală a sistemului - green threads în Clojure sau procesele Erlang). Tot în treacăt amintim şi câteva mecanisme specifice (*pmap* în clojure, *actor model* în Erlang, *goroutines* în Go).

Mai multă atenţie acordăm în schimb modalitătilor de gestiune a stării, cărora mecanismele de concurenţă le sunt auxiliare.

Începem prin sublinierea metodelor de gestiune a memoriei comune grupate în abordarea *Software Transactional Memory*. Sub această umbrelă plasăm atomii, referinţele (Clojure) şi tranzacţiile atomice (Haskell). Primele două se încadrează la gestiunea stării în limbajele funcţionale impure şi reprezintă mecanisme similare *mutex*-urilor (precum *lock*-urile), cu excepţia că gestiunea lor este automată şi prezintă garanţii de siguranţă şi cu diferenţa dată de emfaza pe modificarea lor, specifică limbajelor funcţionale. Tranzacţiile atomice din Haskell sunt similare referinţelor din Clojure (mai multe operaţii cu garanţii de atomicitate), dar sunt aplicate într-un limbaj pur (chestiune de implementare), sub orice alt aspect ele funcţionând de o manieră similară.

Încheiem prezentarea mecanismelor de sincronizare a stării prin prezentarea unui model foarte viabil (şi dovedit în timp) de realizare a concurenţei şi anume a modelului de concurenţă prin actori. Actorii sunt elemente computaţionale corespunzătoare unui fir de procesare care pasează starea şi prelucrarea acesteia între ei sub formă de mesaje, concept subsumat motto-ului "*Nu comunica prin partajarea stării, partajează starea prin comunicare*". Introducem una dintre cele mai de succes implementări ale acestui model (succes datorat în mare parte şi unor alte inovaţii în limbaj) - implementarea Erlang. Erlang introduce conceptul de supervizare şi câteva şabloane de programare concurentă bazate pe threaduri minimaliste virtuale (generate de maşina virtuală) numite procese care comunică starea între ele prin pasare de mesaje (*message passing*). Sistemul este rezistent la erori şi foarte robust în general, fapt ce îl face utilizat în zone în care paralelizarea intensivă este critică (telecom).

Concluzii

Concluzionăm abordarea noastră a concurenţei arătând că necesităţile de procesare curente ne presează înspre o înţelegere mai bună a mecanismelor care stau la baza elaborării programelor care ne automatizează viaţa de zi cu zi. Deşi abordarea funcţională este mai puţin la îndemână întrucât necesită o curbă de învăţare mai puţin accelerată şi chiar dacă productivitatea nu este neapărat cea mai bună în acest caz, această abordare duce la programe mai simple şi mult mai corecte cu necesităţi de mentenanţă mai scăzute.

Credem, de asemenea, că lucrarea contribuie la o înţelegere iniţială a uneltelor aflate la dispoziţie pentru structurarea programelor în vederea concurenţei, lăsând în acelaşi timp loc pentru dezvoltări ulterioare nu doar în înţelegerea celor mai bune practici în software development ci şi în înţelegerea mai bună a realităţii conceptuale.

The live world

Tweet that

We live in a world where everything happens at once. Due to technology advancements we can and like to share as much information as possible about the events in our lives. As a consequence, it is not enough anymore to just share the information, but this information needs to be shared in real time. We want to be the first to break the news about an important event and this phenomenon has evolved into a measure of our social connectivity: we are better if we can give more accurate information in a shorter time.

Dave Lester[1] from Twitter gave a speech at OSOM about the challenges that the company faces at major sports events in the US. In a talk about Apache Mesos he pointed out that a score by one of the teams in the game would trigger a spike in traffic on the platform of over 100 times the normal, which required tremendous technical effort to accommodate. And this is only the microblogging world. In finances, things are even worse because of the high stakes involved.

A world for the impatient

In 2014 a huge scandal broke out[2] involving major banks and high frequency trading on the New York stock market. Apparently banks were investing in software capable of executing big pools of transactions at high speeds and took advantage of the distortion created by this software in the market, by selling preferential access to paying customers. Everyone wanted to trade faster, to seize the opportunities in the market, but not everyone could afford it. Impatience was prompted not only by subjective impulses, but by the very objective financial motivators.

At the same time, we have gotten used to having everything delivered immediately, from information to Amazon packages[3], from weather updates (specific down to the hour) to office lunches. Businesses that do not accommodate our new-found style of living do not stand a chance. And we are only speaking about fine-grained improvements to our lives, but there are areas where real-time updates are critical, like sensitive processes monitoring

[1] http://osom.ro/2014/10/osom-2014-schedule/

[2] Steven Pearlstein,*'Flash Boys': Michael Lewis does it again* - Washington Post article on Michael Lewis' book, April 2014, permalink: http://www.washingtonpost.com/business/flash-boys-michael-lewis-does-it-again/2014/04/12/4a53daf8-bf5d-11e3-b195-dd0c1174052c_story.html

[3] *Amazon Prime Air*, drone package delivery - http://www.amazon.com/b?ie=UTF8&node=8037720011

(nuclear plants), emergency situations (disaster notification systems) or location services (see recent earthquake in Nepal where Google gave the authorities access to their Person Finder service to help locate survivors[4]).

The information infrastructure that is required by such a need for real time poses significant challenges. Some of the challenges were anticipated, some were not, but the computer science community is in agreement that the problems that we face today require new tools or approaches to using the old ones.

In this paper we take a look at two software development approaches and analyze their pros and cons with regards to concurrency. To be noted that we make the same distinction between concurrency and parallelism that most of the papers make[5] and that we are interested mostly in concurrency because of the shift in thought that it imposes.

[4] Sam Frizell, *Google Deploys Person-Finder Tool to Help Survivors of Nepal Earthquake*, time.com article, http://time.com/3835665/nepal-earthquake-relief-google-person-finder/
[5] "Concurrency is not parallelism" - Andrew Gerrand, the Go programming langauge official blog, http://blog.golang.org/concurrency-is-not-parallelism

Object oriented concurrency

Object oriented programming, abbreviated OOP, is a subset of the imperative programming paradigm, where data structures and the functions that operate on them are grouped together in a structure called class that defines a template for specific instances of the class built at runtime and called objects[6]. In essence, objects are supposed to model real world perceived entities and the interactions between them, while maintaining the properties of generality and specificity through abstraction[7]. OOP also defines the concept of encapsulation where access to data is only possible through well controlled interfaces.

Threads, processes, mutability

Concurrency refers to the process of running a program in logically separated threads of execution. The program is composed of independent operations that are logically performed in parallel and that might or might not communicate among them, but they contribute to the overall purpose of the program. At hardware level, logical parallelism might not reflect in actual parallel execution, or any speed improvements for that matter, because of complexities due to operating system's scheduler, or multi-core or multi-machine processing. Nevertheless, concurrency is important because it introduces a framework for thinking about things that happen at the same time.

In OOP concurrency is achieved either through separate processes, or, more commonly, through OS level or virtual machine level threads. Processes are more computationally expensive, so threads that run in a single process or in a pool of processes are more commonly used. Another important distinction between the two is that processes run in a different memory space, while threads, being started by the same process, run in the same memory space, making memory sharing a lot easier.

In OOP, also as a property of imperative programming, variables are mutable, making it transparent for the user to replace the contents of a memory area. Communication between threads is done using such variables, which makes it difficult to know the state of a current variable at any time. This can cause serious problems if multiple threads need to modify the same variable over time, and if the subsequent values are dependent on the previous values.

[6] Stoyan Stefanov, Kumar Chetan Sharma, *Object-Oriented JavaScript Second Edition*, Packt Publishing, 2013, page 12
[7] Alan C. Kay, *The Early History of Smalltalk*, Apple Computer, page 4, http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html

Hold your horses: locks and semaphores

To overcome the problems associated with updating the same mutable variable through different threads, each thread can acquire a lock on the specific variable, perform its operations and then release the lock. All concurrent threads will loop until the lock is released on a variable that they want to update. Note that locks are necessary on both reads and writes, or on a series of read-writes (one read-write, performed for the purpose of updating a variable depending on its previous state is called read-update and it always requires a lock, because no other write operation can interfere between the write and the read that it is dependent on).

The sample Python code[8] uses the Python multiprocessing module to read sensor data from a sensor grid managed by a Sena ProbeeZE10[9] controller attached to the serial port, display a web dashboard for it and push the live data to the web interface via Server Sent Events[10]:

```
from multiprocessing import Process, Lock, Manager,
freeze_support
from http import server
from serverhandlers import DataServerHandler,
CommandServerHandler
import argparse
#import json

def readProcess(l, data, command, databaseName):
    from ProBeeZe10 import ProBeeZe10
    from ProBeeZe10 import ProBeeDatabase
    from time import time

    try:
        device = ProBeeZe10.ProBeeZe10('COM5')
        database =
ProBeeDatabase.ProbeeDatabase(databaseName.value, 500)
        #database.createDeviceTables
        '''
        # AT+REMOTE=0001950000000280,AT+DIO=111111111111
        '''
        device.setRegister(11, 1)
```

[8] Adrian Rosian, *Sena ProBeeZE10 read code with PySerial*, https://github.com/adrianrosian/sena
[9] Sena ProBeeZE10 presentation page, http://www.buysenaindustrial.com/ze10s-00.html
[10] Mozilla Developer Network, *Using server-sent events*, https://developer.mozilla.org/en-US/docs/Server-sent_events/Using_server-sent_events

```
        database.createDeviceTables()
        while True:
            if (command._callmethod('__len__') and
                    command[len(command) - 1] == 'exit'):
                l.acquire()
                print('Read process exiting..')
                l.release()
                break
            line = device.getLine()
            if not line:
                continue
            address, digital, analog = line.split('|')
            address = address.lstrip('+')
            samplingTime = int(time())
            t = samplingTime, analog, digital, address
            database.insertDataSample(t)
            digital = [int(i) if i.isalnum() else None for i
in digital]
            analog = [device.getVoltage(int(i, 16)) if
i.isalnum() else None for i in analog.split(',')]
            data.append({'nodeId': address, 'time':
samplingTime,
                         'analogValue': analog,
'digitalValue': digital})
    except Exception as e:
        l.acquire()
        print('\nAn error has occured: ', type(e), e,
              '\nRead process exiting..')
        l.release()

def serverProcess(l, data, command):
    Handler = DataServerHandler
    httpd = server.HTTPServer(("", 5000), Handler)
    httpd.timeout = 10

    while True:
        if (command._callmethod('__len__') and
                command[len(command) - 1] == 'exit'):
            break
        if data._callmethod('__len__'):
            while data._callmethod('__len__'):
                Handler.data.append(data.pop())
        httpd.handle_request()
        Handler.data = []

if __name__ == '__main__':
    freeze_support()

    # Parse command line arguments
    parser = argparse.ArgumentParser(description='Data server
and controller'
                                     + 'for the Sena ProBee
ZE10 module')
    parser.add_argument('database', metavar='DbName',
type=str,
                   help='SQlite3 database absolute location')
    parser.add_argument('--key', '-key', metavar='accessKey',
type=str,
                   help='Security token to grant access to
device commands')
    args = parser.parse_args()

    # Set the lock and the manager for the shared memory
    lock = Lock()
    manager = Manager()

    # Define shared memory
    sensorData = manager.list()
    command = manager.list()
    databaseName = manager.Value('u', args.database)

    # Start processes
    readProc = Process(target=readProcess, args=(lock,
sensorData, command,

databaseName))
    readProc.start()
    serverProc = Process(target=serverProcess, args=(lock,
sensorData, command))
    serverProc.start()

    # Handle received commands
    allowedCommands = ['exit']
    commandHandler = CommandServerHandler
```

```
        listener = server.HTTPServer(("", 5001), commandHandler)
        listener.timeout = 10

        while True:
            if (command._callmethod('__len__') and
                    command[len(command) - 1] == 'exit'):
                break
            listener.handle_request()
            if hasattr(listener, 'lastCommand') and
    listener.lastCommand:
                print('Command received: ', listener.lastCommand)
                if listener.lastCommand in allowedCommands:
                    command.append(listener.lastCommand)
        readProc.join()
        serverProc.join()

        print('Upon exit, the data list had a size of: ',
              sensorData._callmethod('__len__'))
        print('Main process exiting..')
```

There are some noticeable points in the code: we are using the multiprocessing module with a data *Manager*[11] to hold and control access to shared data amongst the processes.

Specifically, in the main function we have a list for the sensor data to be transmitted between the reading process (the process that reads from the sensor and places the data in the database) and the SSE web server (the process that sends the live data to the web interface, where it is displayed in real-time) - `sensorData = manager.list()` - a list for the commands to be sent to the web server interface and to the read process and the database name, to be sent from the main process to the read process and collected from the main function command-line arguments.

Also, we pass on a lock to the read process to use it for printing on STDOUT in case an exception occurs in the reading process. The lock is necessary as the main process also has access to STDOUT and without it, processes could write characters alternatively to the console, resulting in unreadable error messages.

The problem with locks is, in this case, the stability of the software because various causes may interfere with the lock release, making the other

[11] Sharing memory between Python processes, https://docs.python.org/3.4/library/multiprocessing.html#sharing-state-between-processes

threads loop indefinitely in a process called deadlock. The problem can be avoided using a timeout, a period of time after which the lock is released automatically if the acquiring process could not complete its task. Using a fixed time out can, in turn, lead to another problem, called a livelock, where all the threads reach the same timeout and thus try to perform the same lock-release cycle, making it impossible for any of them to progress. Both deadlocks and livelocks are particular cases of a general problem called resource starvation, where neither thread can progress due to lack of access to necessary resources.

Even though they are run in a logical parallelism, portions of the threads might require serial access to a specific resource (for instance, pieces of an engine can be built independently but the engine must be put together in a specific order). For this particular case a simple lock is not sufficient, and a sequence of locks is necessary, in a construction known as a semaphore[12]. Semaphores are even more complicated to handle because they pose all the problems of a simple lock and also problems associated with the order of operations.

Read-update and atomicity

As previously mentioned, one read followed immediately by a write is called a read-update process. This read-update process is important in terms of concurrency because it is the fundamental place where cross-thread problems can occur. If we imagine the case of multiple threads trying to update a counter of some sort (for instance in the case of collecting content from websites as part of a search engine, we would like to know how many pages were visited to determine progress, with each thread incrementing the number of pages as they complete visiting it), then the operation of incrementing is a two-step operation: we must read the current value of the counter variable in order to write back the incremented value. If this happens in a multiple thread environment, then another thread can do a write to the counter between the read and the update, making the value that we got in the first thread obsolete.

This occurs because the read-update operation is not *atomic*. As read and writes are basic operations that cannot be subdivided, we call them atomic operations. In order to insure the same properties in a read-update cycle we need a locking mechanism like the one described above.

[12] We are referring here to **binary semaphores**, not to **counting semaphores**, which are a variety of semaphore who only count the number of resources available (see the dining philosophers problem with a waiter that doesn’t sit more philosophers at the table than forks available)

The following program[13] in Go performs an incrementation of a shared variable by two independent functions (*goroutines*) with the help of a *Mutex* to ensure a one-step (atomic) incrementation. Functions are part of a *WaitGroup* which forces the main function to wait for the incrementations to complete before continuing:

```
package main
import (
     "fmt"
     "sync"
     "time"
)

var value int
var m sync.Mutex
var wg sync.WaitGroup

func slowInc() {
     m.Lock()
     v := value
     time.Sleep(1e8)
     value = v+1
     m.Unlock()
     wg.Done()
}

func main() {
     wg.Add(2)
     go slowInc()
     go slowInc()
     wg.Wait()
```

[13] Rosetta Code Mutex example, http://rosettacode.org/wiki/Mutex#Deadlock_prevention

```
		fmt.Println(value)

	}
```

Most OOP languages offer libraries or sets of libraries that deal with multithreading and in most cases they have mechanisms to insure atomic updates for variables declared as such. There are, of course, other mechanisms to deal with problems associated with concurrent code, but they are inspired from functional programming languages and we will mention them at the proper time.

Complexity and its sources

There have been many studies on the causes of complexity in computer programs. Large software is often hard to maintain and any progress in reducing the maintenance cost associated with large codebases is welcome. The obvious cost for complexity is, thus, the sheer number of lines of code. However, that cannot be the only criterion. Another obvious answer would be different programming styles of different programmers, which translates different ways of approaching software development into code complexity. But even so, not all complexity can be accounted for. So what is the main cause of complexity in software?

Stating the not-so-obvious

Complexity can be categorized from the start into two main types: intrinsic complexity deriving from the complexity of the business model that we are trying to emulate and accidental complexity, which has its source in the tooling used to build the software[14].

Business model complexity[15] cannot be avoided, at least not without altering the business model that it is based on. You cannot design software that goes against the normal workflows of a business process without disrupting the activity. Business model complexity must be acknowledged and grouped into the code in such a way that it is fairly easy to modify in case of changes in the underlying business flows. This is the reason for most of the abstractions provided by programming languages: to allow easy code refactorings that reflect changes in the workflows or business data.

Accidental complexity[16] refers to complexity introduced by the tools and programming languages used to create the software. Given the design and the purpose of a specific language or tool, it might require a varying degree of housekeeping to adjust it to a specific use case. It is easier to design some specific software in a specific language, rather than in others. This is the source of a good software architect's mantra "*use the right tool for the job*" or, the semantic opposite "if all you have is a hammer, everything looks like a nail".

More generally, the complexity caused by the tooling can be filed under the category of complexity caused by control. Tools that force you to choose a certain path in achieving a programming goal introduce artificial control flows

[14] Ben Moseley, Peter Marks, *Out of the Tar Pit,* February 6, 2006, page 5 - Causes of Complexity
[15] Mosely, Marks, Op. cit. p.21 - Accidents and Essence
[16] Ibid.

in your program. They do not serve the overall purpose of the program, but are only necessary because there is simply no other way to move forward.

A special case of control[17] related issues appears in imperative programming. As previously stated, in imperative programming one usually deals with mutable state. To manage mutable states imperative programming offers control structures that tell the computer how to achieve a specific purpose, as opposed to what to achieve. This is most of the time unnecessary as mathematics proves to us. We can mathematically describe a system without necessarily describing all the transformation steps necessary to get from one state to another. Take for instance a for loop, present in most imperative programming languages: it usually consists of a counter variable that increments on every step and a block that is executed until the counter variable reaches a specific limit. This assumes that there is a sequence in which the steps need to be performed, but most of the uses of a for loop are for its side-effects (executing the inner block a number of times), not for this assumed order of events. Nevertheless, in a for loop we have, at every step, a state of the counter, which is not always very predictable, depending on the way the for loop is built. The problem exacerbates as the number of such for loops grows in a program.

We can observe that the underlying problem here seems to be a programmer's ability to predict the state in which the program is at all times. This is the reason for the existence of software debuggers, that allow you to trace every step of the execution of a program and to inspect the state of any mutable variable used along the way. State seems to be the main source of errors that are not related to business logic, so the source of all accidental complexity.

Not a role model

We can go even further and ask ourselves: if state is the source of all accidental complexity, what is the source of state? Is there any state inside a business model? Is business model state the source of business model complexity? In other words, is state also behind logical software errors, is the model that we have now flawed?

These are bold statements and require a more detailed inspection of what state is. This will take us down a philosophical path towards what reality is, but the journey is worth the effort because it seems to be consistent with most cases where software errors appear.

[17] Ibid.

Kant[18] was one of the greatest philosophers of modern times because he seems to have settled, at least in the author's narrow understanding, the old dispute between subjective knowledge and objective knowledge acquisition, as described in the theories of Schopenhauer and Leibniz.

In Schopenhauer's view[19], starting from Descartes' affirmation that *Dubito ergo cogito, cogito ergo sum*, we can only say that the only thing that we know for sure is that we exist as a person because we can attach an action to an entity (*cogito* is the action that we can attach to our person) and any further perception is subjective because it is acquired through our senses and we cannot get outside the limit of our senses to see how reality is.

Starting from the same assertion of Descartes, Leibniz reaches the conclusion that objective knowledge is possible through mathematics. Descartes' affirmation is proof of the mathematical unity (the entity to which we attach the action of *cogito*) and the contrary is proof of non-existence, or of the digit 0. By making use of these building blocks, true, false and one and zero, one can build the entire abstraction of mathematics, based on which we can even construct logic assertions that are always true because they are mathematically provable.

Kant shows that neither Schopenhauer's theory, nor Leibniz' theory is sufficient to describe the reality and unifies them by identifying twelve so called *categories* representing maximum generality terms with the help of which one can construct objectively through assertions and define subjectivity as anything that falls outside of these categories. In consequence, reality is much more than what is available to us through our senses; nonetheless there are aspects of reality that can be objectively described.

This is consistent with the processual-organic[20] view of the world: human beings possess a special type of bio-processors that are capable to alter themselves under the pressure of necessities, called *interpreters*. Reality is composed of info-energy, with information having the role of organizing energy and energy having the role of supporting the existence of information. In this context, the necessities that pressure the human interpreters to evolve are created by interaction with info-energy that cannot be processed by the current state of the interpreters.

To clarify what this has to do with state, we will quote the SICP[21] book, which has been the reference programming manual for MIT for a long number of years, which describes a graffiti on a university wall: "Time is a machine

[18] Sir Roger Scruton, *Kant*, Humanitas 2006
[19] Arthur Schopenhauer, *Lumea ca voinţă şi reprezentare*, Humanitas 2012
[20] Lucian Culda, *Procesualitatea sociala*, Licorna 1994
[21] Harold Abelson, Gerald Jay Sussman, Julie Sussman, *Structure and interpretation of computer programs, 2nd edition*, The MIT Press, 1996

invented so that all things don't happen at once”. It would seem that state is a result of our inability to perceive reality as is, resulting in the concept of time. Each piece of information that we receive alters our interpreters, allowing us to perceive additional information, which creates the illusion of a changing state and thus, of time.

Under this new model, state can be eliminated from computer programs by closely following a more mathematical model that deals with accidental complexity and confines as much as possible the business model complexity.

Functional concurrency

Functional programming draws its name from the approach used in designing programs which are viewed as one main function that calls internal functions to achieve its goal. Functions are nested down to the level of language primitives, that are normally very simple and composable into more complex data types.

Functional programming theorists argue that because of its closeness to mathematics functional programming is much more likely to give correct programs that are easily testable and that are flexible without increasing complexity.

Assuming this is true, functional languages seem the ideal candidate for concurrency oriented programs. The problem in proving this point is the lack of a unifying framework in comparing the two approaches. Although fairly close to this, complexity seems to be only part of the story. The remainder of this picture seems to be situated in the eternal battle between individual affinities to certain programming languages or paradigms, or between wounded egos.

Because of the properties of functional programming languages, functional programming theorists argue that concurrency (especially in pure functional languages) is merely a way to organize effectless code (code without side effects) to better handle the interaction between multiple independent entities and to avoid the so called *spaghetti code* generated by callbacks or evented code.

Functional languages use for concurrency much of the same approaches from imperative languages (mostly because threads and processes are operating system constructs), with some notable exceptions (Erlang processes are, like the new Java fibers, lightweight processes, comparable to threads, with the advantage that the memory space is separated, since they are handled by the Erlang VM[22]). However, the emphasis on safety dictates some interesting solutions: we already mentioned lightweight processes, but notable approaches are also futures, parallel mappers/reducers and their async mechanisms - promises and delays. Another approach that is equally interesting, although not specific to functional languages, is based on concurrent coroutines and their synchronization mechanism - channels[23].

Immutability and purity

Functional programming languages claim their spot at the top of the list based on a set of attributes that give them the elegance of mathematical equations. In order to achieve this, the programs written in these languages must have the properties of these equations.

In many of the functional programming languages this property is part of a protocol (or interface) called *Equatable*[24] that basically ensures that two data types can be compared. But there is one impediment in achieving this in imperative programming, and that is the use of the equal sign (=). In mathematics this symbol has a very clear meaning, and it is used to designate

[22] Erlang processes, http://www.erlang.org/doc/efficiency_guide/processes.html
[23] This is true for Clojure *core.async* as well as for the imperative Google language *Go*
[24] Miran Lipovača, *Learn You a Haskell for Great Good! A beginner’s guide*, No Starch Press, April 2011 - Type classes 101

two quantities that are deemed identical. Imperative programming uses the symbol to designate attribution, in which the left operand is assigned the value of the right side, basically mutating this value. Mutability is a fundamental characteristic of imperative programming and is linked to state management and also to the flow of the program.

To correct this, functional programming uses the concept of immutability. If one changes the value of a variable, it actually transforms it into something else and the use of the equal sign to do this would be incorrect because the properties of Equatable wouldn't hold true anymore. So the transition from the initial user state of the program to its result is done by a series of functions that take as input immutable parameters and return results that are used as immutable parameters for the encompassing function. Even though in von Neumman machines this approach is still translated into mutating memory locations situated at a limited set of addresses, which might mean that more than once it would actually be a replacement of the same memory area, this approach is transparent to the programmer and it guarantees that there are no side effects to a function execution with regards to its parameters.

Assignment (improperly used here) is done in functional programming through binding and unwrapping, which are subsets of the properties defined in Equatable, which doesn't make it an actual assignment but an identity linking between two equal structures that holds the mathematical properties of the equal sign.

Of course, no program would make sense without state management, but depending on where this state is managed, functional languages are classified in *pure* and *impure* languages.

In a *pure* language, state is handled in a very specific data structure that encompasses a value and its context and that can be passed along as an immutable parameter that can be unwrapped into its two components for the purpose of processing them separately and composing a new structure of the same type. Such a structure is called a Monad. A Monad can pass along its context through a function pipeline, from the user input through the program result, allowing functions along the way to have no side effects (same input parameters would always yield the same result) and thus remain pure.

The typeclass for the Monad construct, in Haskell, looks like this[25]:

```
class Monad m where
    return :: a -> m a

    (>>=) :: m a -> (a -> m b) -> m b

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail msg = error msg
```

This is a parametrized type class, that takes a simple data type *a* and wraps it in a the context specific to the monad. Of interest to us in the

[25] Lipovača, Op.cit., The Monad type class

construction of the Monad here is the *return* function, which wraps the type *a* with the monad context *m* and the *bind (>>=)* function, which tells us how we transform a value with context (Monad) through a function.

Below is the instantiation of the type class as the monad *Maybe*. *Maybe* is a value with a context of failure. It represents the value itself (*Just a*) if there is a value, or *Nothing* if there is no value inside the type. This is useful for passing the context of error along the way in the functions and it compels the programmer to take into account the errors as values (so that errors themselves can be addressed as function parameters):

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f  = f x
    fail _ = Nothing
```

We see the way that functions are *bound* in the case of this monad: any function applied to a non-existing value will return *Nothing*, while applying the function to some existing value makes sense and constitutes the normal operation of the function *f*.

If we want to define a function *add* that operates on *Maybe* types, it might look like this[26]:

```
add :: Maybe Int -> Maybe Int -> Maybe Int
add mx my =
  case mx of
    Nothing -> Nothing
    Just x  -> case my of
                 Nothing -> Nothing
                 Just y  -> Just (x + y)
```

The function returns a value only when both *Maybe*s are *Just a*, otherwise it returns *Nothing*. This code won't even compile if the values passed to the function don't match the types and it treats the possibility for failure gracefully, as the return value of the function. One would argue that this is similar to a multiple return value in any programming language, but the subtle difference is that the data type itself compels the programmer to account for the possible failure, as opposed to a runtime check.

In an *impure* language, state can be handled at any place in the program with minimal formalities, but nevertheless with the use of markers to identify the places where this happens. Both *pure* and *impure* languages treat state as the main possible source of errors and act accordingly.

Composability and abstraction

Most of the criticism targeted at imperative programs underlines their incredible verbosity when trying to extend them or in any way compose modules written using this approach. Program design patterns that are used by imperative languages to address issues of reusability are complex and difficult

[26] Bryan O'Sullivan, John Goerzen, Don Stewart, *Real World Haskell*, O'Reilly 2009, chapter 14

to master for beginners and their misuse is consequently a major source of error in software development.

Knowing that the main source of complexity in software is state management, it is easy to understand why software patterns like Abstract Factory or Observable can easily lead to complex code and to misuse. The programmer must understand unequivocally the rules of inheritance, overriding, polymorphism and encapsulation to correctly manage the interfaces, classes and the resulting objects along with their treatment of mutable state. On top of these rules, the artificially induced control flow is increasing the complexity of the code through hard to follow loops and complex mutating conditions for the state variables.

Since code reusability is something very desired in the software industry because of numerous advantages, composability and abstraction are features that are highly desirable in any programming language.

Because of their seemingly mathematical structure, functional programming languages benefit for free of the mathematical rules for function composition. Since in functional languages functions are first class citizens (they can be passed as parameters to other functions and can be returned from other functions) and since they can be partially applied, most of the rules of functional composition in mathematics are already met.

This gives birth to interesting properties in functional programming languages, such as complete lack of loops (which are only provided in some languages as syntactic sugar) and complete freedom with respect to order of execution (with the notable exception of portions where state management is done).

Consider the following two functions, used to convert a list of *1*s and *0*s into their binary representation to be written in a file:

```
(defn bits-to-byte [byt]
     (letfn [(bit-red [acc el]
          (+ (bit-shift-left (int acc) 1) (int el)))]
          (reduce  bit-red byt)))

(defn bit-list-bytes [bits]
     (map bits-to-byte (partition-all 8 bits)))
```

The function *bit-list-bytes* maps the function *bits-to-byte* (applies to every element) over all 8 element partitions of the bit list. This second function *reduces* (applies a function that transforms a list into one element) the 8 bit list to a single element (a byte value, or a small integer) by applying the internally defined function *bit-red*, which obtains the byte value by adding the left-shifted value of the former’s bit integer representation to the integer representation of the new bit.

Notice how functions can be used globally (not recommended, as it is less flexible - like in the usage of *bits-to-byte* inside *bit-list-bytes* without the mapping function being passed as param) or can be passed on as parameters (like in the usage of the internally-defined *bit-red* as a parameter for *reduce*).

Also please note that, since there are no loops involved, other than the functional dependency, there is no order of operations that is implicit to the program - the *map* and the *reduce* functions are being applied, as opposed to following a control path (like in a loop, where steps are incrementally taken to achieve the goal).

At the same time, since functions represent general rules of data transformation (as opposed to data processing in imperative languages), they can be evaluated only when this is needed in a process called lazy evaluation. Laziness can, due to this property, generate the interesting concept of streams (or infinite lists), from which values are only *realized* when needed.

The nuclear solution: atoms and STM

Even though most functional languages are immutable, impure functional languages allow some level of mutability. The difference between functional languages and imperative languages is that the focus, or the emphasis, is on the immutability of things. Data is composed of values, values do not change. In functional languages one dissociates identity from value and is fully aware that a new value means a different identity. If one changes a value, it switches identity at the same time and s/he is fully aware of the fact.

We encounter one such example of controlled mutability in the case of the Clojure concurrent primitive called *atom*[27]. The concept was so useful that even some OO languages adopted it. An atom is a language construct that allows one to bind the same identity to different values over time. The process of getting the current value bound to the identity is called a *deref* (from *de-referencing*) and the process of updating the value is called a *swap*. It is important to notice that a swap is atomic, it is a one-step operation from the point of view of any other segment of the program trying to deref the atom. Until the swap is complete, any deref will bind to the former stale value of the atom, which prevents race conditions in a concurrent program.

We can use an atom to synchronize the state across multiple threads started from futures. In a chat program, for instance, each conversation would happen in a separate thread to avoid blocking the server. Threads need to be aware of their socket connection and of the other thread's socket connection so they can pass messages between sockets (in a chat room, for instance). All the socket connections can be a part of an atomic list where the connections are added and removed as needed using swap and messages are passed from one socket to the others by mapping a function over the list.

```
(defonce channels (atom #{}))
(defn handle-new-connection [channel]
     (info "New connection received -" channel)
     (swap! channels conj channel)
     (send-data channel “Welcome”))

(defn connect-to-chat [req]
    (with-channel req channel
      (handle-new-connection channel)
      (on-close channel (fn [status]
          (info “User logged out”)))
      (on-receive channel (fn [data]
          (future (handle-data-on-socket data channel))))))
```

[27] Paul Butcher, *Seven Concurrency Models in Seven Weeks When Threads Unravel*, The Pragmatic Programmers, July 2014, chapter 4, p.85

In the above example, the *channels* identity is bound to an atom that gets updated with the new connection information using the *conj*(oin) function. The new channel is then used to send a welcome message to the connecting user. Information in the *channels* atom is used to be able to later on send messages to the participants (see the welcome message, sent on the channel with *send-data*). In the *connect-to-chat* method we can also notice the use of futures for dispatching the handling of received messages (see the handlers for the *on-receive* function call, passed as a second parameter lambda-function).

As synchronization primitives, atoms can be considered a particular case of an approach called Software Transactional Memory[28], specifically of *refs*. Much like database transactions, refs allow synchronized and atomic modifications of the state of multiple identities. Same as with atoms, states can be validated and failure to validate of any of the updates will cause the entire transaction to fail (much like transaction rollback in RDBMS).

```
(def acc-bob (ref 100))
(def acc-sam (ref 300))

(defn transfer [from to amount]
  (dosync
    (alter from - amount)
    (alter to + amount)))

(transfer acc-bob acc-sam 10)
```

In the above example, we want to make sure that a bank transfer between Sam and Bob happens atomically, irrespective of how many concurrent transactions are happening, so we enclose the calls to ref's *alter* functions in a transaction using *dosync*. The *alter* function applies to the first of its parameters the function received as the second parameter with the argument received as the third parameter. If any of the *alter* function fails for some reason, the whole operation fails to produce side effects and the accounts are not modified, which is what we intended.

At the same time, if multiple concurrent threads try to make conflicting changes to the same values in the transaction, the transaction will be retried. At the end of a transaction, either all modifications in the transaction can be performed (no other thread is trying to do the same thing) or they will be restarted to be performed later.

Same example in Haskell shows the same properties of STM[29]:

```
--file bank.hs
import Control.Concurrent.STM
import Control.Monad

newtype USD = USD Int
```

[28] Butcher, op. cit., p.97

[29] The Control.Concurrent package documentation, http://hackage.haskell.org/package/stm-2.2.0.1/docs/Control-Concurrent-STM.html

```
    deriving (Eq, Ord, Show, Num)

type Acct = TVar USD

data Holder = Holder {
      balance :: Acct
}

basicTransfer qty fromBal toBal = do
  fromQty <- readTVar fromBal
  toQty   <- readTVar toBal
  writeTVar fromBal (fromQty - qty)
  writeTVar toBal   (toQty + qty)

transfer = do
  bob <- newTVar (10 :: USD)
  sam   <- newTVar 5
  basicTransfer 3 bob sam
  liftM2 (,) (readTVar bob) (readTVar sam)
```

We transfer money from one account to the other using the *atomically* call, like this, in *ghci (the Glasgow Haskell Compiler)*:

```
Prelude> :load bank
[1 of 1] Compiling Main             ( bank.hs, interpreted )
Ok, modules loaded: Main.
*Main> atomically transfer
Loading package array-0.5.0.0 ... linking ... done.
Loading package stm-2.4.4 ... linking ... done.
(USD 7,USD 8)
```

The *TVar* type is a parameterized type (a type that receives another type *a* as parameter and returns a type wrapper around it - in our case the type *a* is *USD*, which is an alias for an integer) and it represents a mutable variable that can be *atomically* updated.

The signature of the *basicTransfer* function, called in the wrapper function *transfer* is *basicTransfer :: Num a => a -> TVar a -> TVar a -> STM ()* , indicating that both *bob* and *sam* are *TVar* variables wrapped around *Holder*. If we were to run the transfer functions from Haskell threads, it would present the same properties as a Clojure ref.

Act like you mean it: share your state

The theory behind the *Actor model* of programming has been around since 1973[30]. On the same principle as the OOP approach, the Actor model states that everything is an actor. The main merit of the model is the decoupling of

[30] Hewitt, Carl, Peter Bishop, and Richard Steiger, *A Universal Modular Actor Formalism for Artificial Intelligence*, In IJCAI-73: THIRD INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 20-23 August 1973, Stanford University Stanford, California. 1973

the message that is sent from one Actor to another one in the context of concurrency (Actors are inherently concurrent).

Arguably one of the most popular early languages that implement the actor model is Erlang. Erlang has been made public in the doctoral thesis of his creator, Joe Armstrong[31] and it presents some characteristics that make it noteworthy from the point of view of reliability of a concurrent system. The system presented in Armstrong's doctoral thesis achieved a *nine nines (0.99..9%, up to nine digits)* reliability score as during 40 years of functioning it had a cumulated downtime of only 2 hours.

In Erlang there is no global state and state is maintained internally under the form of messages passed around between lightweight processes. This alleviates most of the problems associated with shared state, as it serializes access to the state (an actor can pass on a modified state - an immutable copy of the original state - only after it receives it under the form of a message).

This is the embodiment of the principle *"Do not communicate by sharing state, share state by communicating"*[32] and it is modeled after the way people in location-distributed organizations collaborate. Let us look at the example of a chat engine's presence service which tracks the online/offline status of participants (much like the Clojure example previously presented):

```
-module(presence_websocket,[Req, SessionId]).
-compile(export_all).

init() ->
    State = [],
    {ok, State}.

% Handles a client joining the server.
handle_join(Url, Pid, State) ->
    UserInfo = login_helper:isLoggedIn(SessionId),
    case UserInfo of
        undefined -> {stop, "Unauthenticated", State};
        Admin -> {noreply, State ++ [Pid]}
    end.

% Handles a client closing the connection.
handle_close(Reason, Url, Pid, State) ->
    NewState = [X || X <- State, X =/= Pid],
    {stop, "Connection closed", NewState}.

% Handles incoming message.
handle_incoming(Url, Pid, Message, State) ->
    {noreply, State}.

handle_info(Info, State) ->
    lager:info("Received info: ~p", [Info]),
    {noreply, State}.
```

[31] Joe Armstrong, *Making reliable distributed systems in the presence of software errors*, doctoral thesis for The Royal Institute of Technology, Sweden 2003
[32] Andrew Gerrand, the Go blog, http://blog.golang.org/share-memory-by-communicating

The module[33] is initialized with a request *Req* and a session identifier *SessionId* by a *gen_server*, a process responsible for starting the individual processes that handle the incoming Websocket connections from the clients (assume a javascript browser client, for this instance). The starting process then initializes the state of the sub-process with its internal state, that is defined at startup by calling the module's *init* function. We can notice that the init function returns an empty list as result, representing no connected clients at the time of startup.

Afterwards, for each connecting client, the parent process calls the *handle_join* function, passing it the *Url* from the request, the *Pid (process id)* of the handling process and the current internal *State* that was initialized at startup. The function returns the new state (new list of clients), obtained from the initial state, concatenated with the current client's *Pid*.

Similarly, closing a connection would trigger the call of the *handle_close* function, which returns a new list of connections obtained from the initial *State* by filtering out of it the current *Pid* (the new state, *NewState,* is formed out of elements *X* that draw from the initial *State* list, with the property that *X* is not equal to the current *Pid*).

Any message sent from the client or from other processes is handled by *handle_incoming* which in this case ignores everything and leaves the state unchanged. There is also some logic for receiving debug info in the last function in the code.

Following the same logic, we would implement the banking calculations from the previous examples like this:

```
-module(banking).
-compile(export_all).

init() ->
    Accounts = [10, 5], %[Bob, Sam]
    {ok, Accounts}.

% Handles a money transfer.
handle_transfer(Amount, Pid, Accounts) ->
    [Bob, Sam] = Accounts,
    [Bob - Amount, Sam + Amount].
```

Along with immutability, functional approach and the philosophy *"Let it crash"* (reduce code quantity by only treating cases for the normal flow of the program and let the exceptional flows crash parts of the system), this property makes Erlang one of the most reliable languages for concurrent systems programming.

The Erlang model has become a source of inspiration for other functional languages as well (*core.async* in Clojure, *Akka* in Scala etc.), but it also inspired some imperative programming languages to adopt the concepts of *coroutines* (functions, or *routines* that execute concurrently with their callers)

[33] The module is using ChicagoBoss, the Erlang framework, particularly the Websocket behavior (which is built over Cowboy's websockets) - http://chicagoboss.org/doc/api-websocket.html

and *channels* (communication primitives, similar to Erlang process inboxes for receiving messages, used to communicate between concurrent executing routines). One such notable example from the imperative world is Go, the language developed by Robert Griesemer, Rob Pike, and Ken Thompson at Google[34].

The same bank account program, written in Go this time will look like this:

```
package main

import "fmt"

type Bank struct {
     bob int
     sam int
}

func transfer(amount int, stateChan chan Bank) {
     bank := <-stateChan
     newBank := Bank{bob: bank.bob - amount, sam: bank.sam +
amount}
     stateChan<-newBank
}

func main() {
     var stateChan chan Bank = make(chan Bank)
     var state Bank
     go transfer(3, stateChan)

     b := Bank{bob: 10, sam: 5}
     stateChan<-b
     state = <-stateChan
     fmt.Println(state)
}
```

We can compile the example above on Linux using the Go language Docker[35] container, to avoid setting up the entire environment:

```
[adrian@localhost   bank]   $   sudo   docker   run   --rm   -v
"$PWD":/usr/src/myapp  -w  /usr/src/myapp  golang:latest  go
build -v
_/usr/src/myapp
[adrian@localhost bank] $ ./myapp
{7 8}
```

The result is, as expected, a successful transfer. In the above code we start by declaring the bank type, then the *transfer* function. The function accepts as parameters the amount and a channel that can conduct *Bank* items.

[34] The Go language, http://golang.org/
[35] Linux LXC wrapper and controller, written in Go - https://www.docker.com/

A channel is similar to the Erlang *Pid* , but they are different in the same way as writing to a file by using its name (Erlang) or a file handle (Go). Channels are like a Unix pipe, but they can also be buffered (can hold multiple values that await draining).

We call the transfer function as a *goroutine* (a Go *coroutine*) from the *main* function and we pass it along the amount and a newly created channel. The goroutine blocks until there is a value on the channel (to get the bank account, we pass it on in the main function using the *stateChan<-b* call). After creating the new state (new bank account value) the goroutine puts the new value on the channel to make it available to the main function (which drains the channel and prints the new result).

We can easily imagine the main function behaving like an Erlang *gen_server* by taking care of scheduling the goroutine calls and passing them on the state in an atomic fashion. For this, we would need the goroutines to behave more like an Erlang process, which can be achieved by creating the communication channel inside the function and returning it for reference, much like an Erlang *Pid*. This is actually the recommended way to create coroutines, according to Rob Pike's presentation on Go concurrency:

```
func transfer(amount int) chan Bank {
     var pid chan Bank = make(chan Bank)
     go func () {
          bank := <-pid
          newBank  :=  Bank{bob:  bank.bob  -  amount,  sam:
bank.sam + amount}
          pid<-newBank
     }
     return pid
}
```

The main function (or a secondary function used as scheduler) would call each transfer, get their *pid* and send on the necessary information as modified state, just like an Erlang scheduler would. The transfer function encloses a goroutine that does the actual work and that blocks waiting for the state to process. When everything is complete, the task would push the result into the *pid* pipe.

Although Go is an imperative programming language, it borrowed a lot from the functional world (closures, functions as first-class citizens). The Go community was wondering why not the rest of the functional characteristics were implemented (maybe currying, tail-call optimizations - like OCaml did, for instance), but the fact remains that Golang was designed as a language for system works at Google, so factors like C interoperability and specific system applications might have weighed in, as well as other considerations unknown to the author, given the experience of the language's designers.

Conclusions

Imperative programming has been around for decades. It has done its job and it did it rather well in a world where software development needs have started to grow. It developed organically out of an early model of the world as we perceived it: objectual, physical, sequential.

Most people can only focus on one thing at the time and we solve our problems that way. In order for us to process reality, we need to take things one at a time because our problem-solving part of the mind imposes that we shift the attention to the problem at hand to start to solve it. We do this in our systems as well: we *modulate* and *demodulate* the electronic signals (think transmissions) to get our news (an actual form of serialization), we do this by imposing rules in our society (oh, wondrous bureaucracy!), we even do this in our language - because aren't we channeling our thoughts into speech and writing, effectively serializing ideas so they are received in the proper order by our conversation partners, in the hope that they will spring in their minds the same thought process that happened in our minds?

Language is an even more interesting case than a shallow look would invest it with. It is, by its evolution, the deposit of part of a people's history - the language itself can tell part of that people's story, as every linguist knows it. We represent through it the world around us, in the process of communication and, by a process that is still a mystery to humankind, we somehow reach a common understanding of our surroundings. The weaknesses of language are obvious in the process of translation, when meaning is lost across language boundaries - "*Traduttore, Traditore!*" is an expression well-known by translators meaning (sic!) "Translator, traitor!".

Even with its weaknesses, we use language to represent the object of our world - we have *nouns* to name objects and phenomena that surrounds us. So, again, it is only natural that we design the computer programs that model our world in the same fashion, by using objects and actions that can be performed by them. And, using the same pattern as in our language, we design sequences of actions to be performed by objects upon each-other to reach a final desired state. Even the term *program* in some languages (like the author's native Romanian) has the meaning *schedule* to express the sequential fashion in which we think of transformations to the data.

For most of the applications this is enough and it allowed us to build massive systems that effectively operate our world. Being embedded in the fabric of the world around us, taking *time* into account (as the source of serialization, or sequential state) has allowed us to scale and develop in all fields of human activity, it has served us well.

Then the theories that have shaken our world started to appear. Einstein first tells us that the perception of time is relative to the speed we travel, then that space and time are interleaved (because gravity is a disturbance in the space-time by objects with large mass). Then comes Heisenberg, with his *uncertainty principle*, about which Einstein famously said "*I am utterly convinced that God does not play dice with the universe*" (Heinsenberg states that observing the phenomenon influences the phenomenon, because the natural state of things is for them to be uncertain, probabilistic - the discussion is about momentum and position in quantum mechanics), triggering an equally famous response from Niels Bohr "*Don't tell God what to do!*". Physicists start to make analogies with Eastern ancient culture[36] and modern physics finds way to translate between dimensions of the universe to unify quantum mechanics and gravity[37].

At the same time with reaching the limits of our model of the world, we reach the limits of our computational model. State, defined in the properties of objects, starts to hinder our calculations, as encapsulation proves to be insufficient to protect us from undesired side-effects. We come to realize that parallel computing is our only hope to speed up calculations, as we reach the limits of Moore's Law. At the level of circuit minification that we have reached, the electrons don't respect the boundaries of the conductors anymore, taking any determinism away from computing. This results in the need for a more rigorous approach to state management until the models that we work with manage to completely factor time out of the equation.

Functional programming shows great promise in this area. While pure functional languages strive to make state an implicit part of the functional mechanism through Monads, impure languages just shift the focus from altering the state at all times to a more controlled approach. The most practically successful approach seems to be the Actor model, which passes state along much like humans do when they collaborate (actually, the roots of the model go way deeper, to the original ideas of quantum mechanics).

Functional programming has the disadvantage of requiring a different mindset, but as it's often the case, change cannot happen unless we approach things differently. There are numerous studies that show that the functional programming perspective might be more suitable for an ever-changing

[36] Fritjof Capra, *The Tao of Physics*, Shambhala Publications, 1975
[37] Ron Cowen, *Simulations back up theory that Universe is a hologram,* Nature article on the Holographic principle discoveries, http://www.nature.com/news/simulations-back-up-theory-that-universe-is-a-hologram-1.14328#/b1

environment and that outline the pros and cons of using them, which can be summarized in a very simplistic way by the statement: *functional languages are more flexible in environments where the functionality applied to the data changes very often, whereas imperative OO languages are more applicable to environments where the structure of the data changes often*. There are, however, typed functional languages that alleviate the problems with the data structure and that maintain the same other nice properties with regards to composability and concurrency.

We conclude by hoping that with a more in-depth understanding of the world that surrounds us and the toolset for knowledge at our disposal, our software will prove to be not only a way to run the devices of our perceived world, but instrumental to our understanding of the reality.