UNIVERSITATEA TEHNICĂ A MOLDOVEI

Real Time Programming

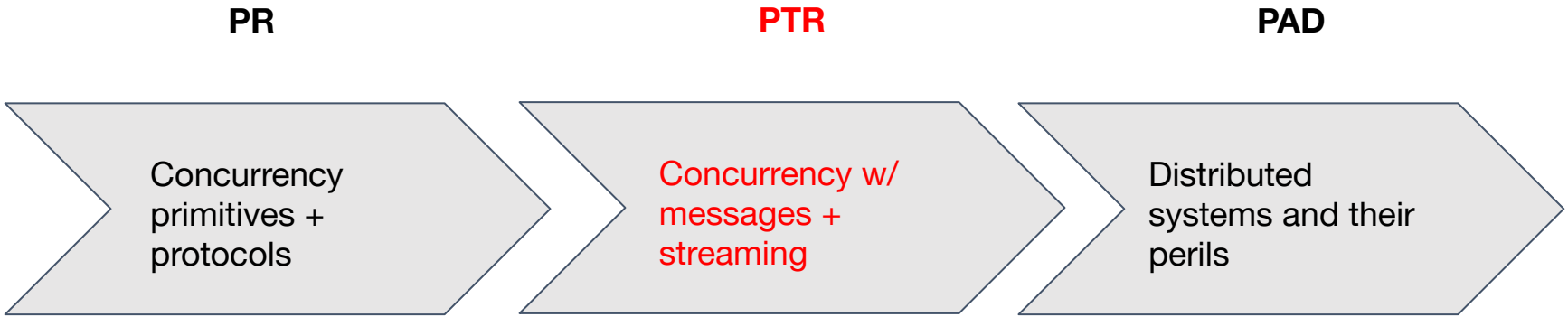
Alexandru Burlacu

Spring 2020

Intro & Course Description

Alexandru Burlacu

Spring 2020


PR
PTR
PAD
Concurrency primitives + protocols
Concurrency w/ messages + streaming
Distributed systems and their perils

- **Topics** - first more advanced Concurrency, and then Message Queues, Streaming, a couple of Protocols
- **Labs** - 1st Concurrency, second Message Queues and real time processing
- **Midterms** - two midterms, a lab (70%) + questions (3 Qs = 5 + 15 + 10)
- **Exam** - oral, 30 min preparation time, <16 min Q&A
- **Grading policy** - 10 is thresholded at 91 points, the rest are relative, following Gaussian dist.
- **Attendance** - Doesn’t matter. Just pass the exam and complete the labs on time

A. Actor model
B. Communicating Sequential Processes
C. Reactive Programming
D. Message Queues
E. MQTT and XMPP
F. Streaming Algorithms

[Higher level]
Concurrency

Alexandru Burlacu

Spring 2020

Many Flavors of Concurrency

Remember it from last semester?

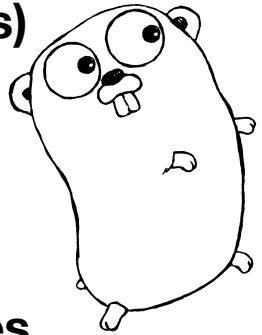
Alexandru Burlacu

Spring 2020

Actors

Event Loop/ Coroutines

CSP (communicating sequential processes)



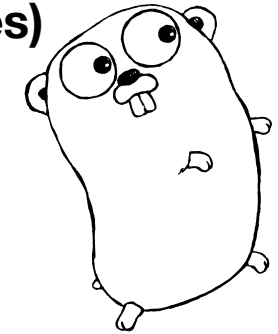
STM (software transactional memory)

Also:

- **Tasks/Futures**
- **Dataflow**
- **Just Threads**

Actors

CSP (communicating sequential processes)



\+ ReactiveX

Actor Model

Alexandru Burlacu

Spring 2020

Actor Model

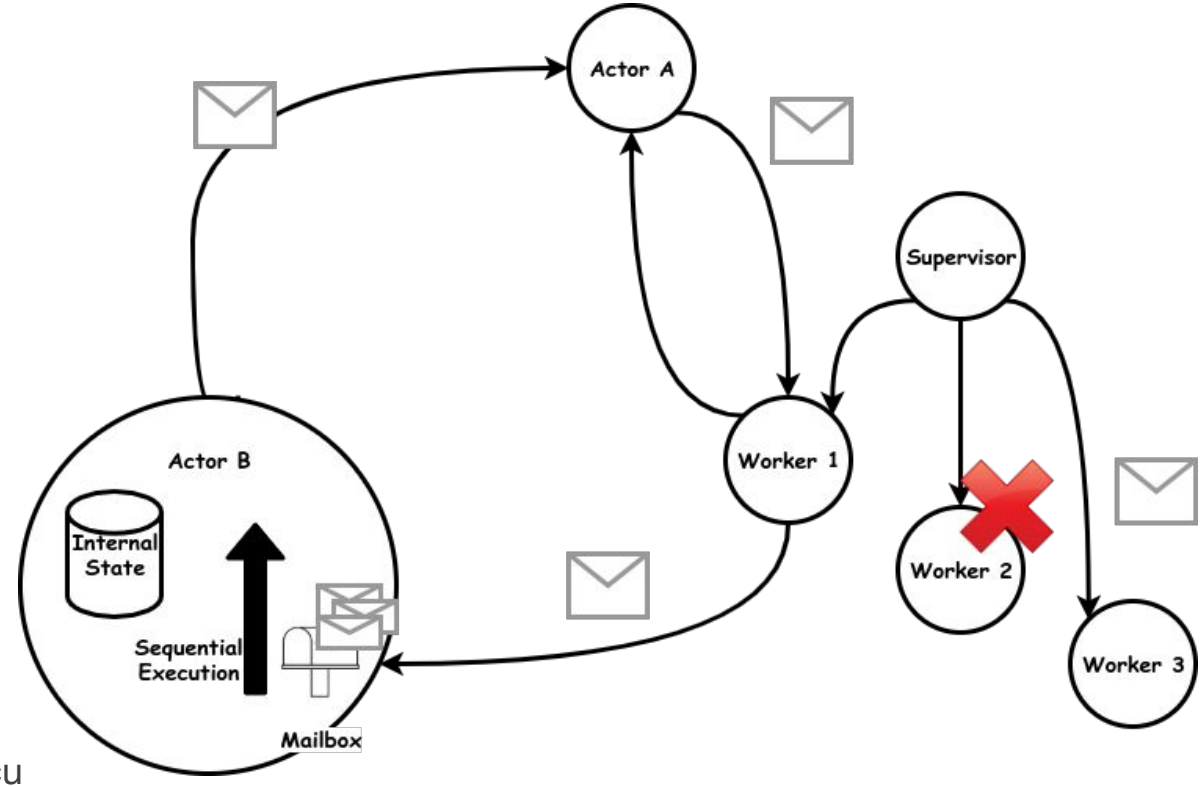
Initially proposed by Carl Hewitt*, Actor Model was a way to structure/model concurrent systems that took into account the perils of shared memory.

Actor Model is a message passing paradigm, inspired from physics and biology, as a result being a bit less formal than for example CSP.

*Carl Hewitt; Peter Bishop & Richard Steiger (1973). "A Universal Modular Actor Formalism for Artificial Intelligence"

One actor is no actor

Alexandru Burlacu


Actor A
Supervisor
Worker 1
Actor B
Internal State
Sequential Execution
Mailbox
Worker 2
Worker 3

One actor is no actor

Actors come in systems

- Actor communicate by sending messages
- Computations inside actors are sequential, but communication gives rise to concurrency within the system
- The state in the system is still mutable, but encapsulated in many entities that are allowed to alter it

Sounds almost like OOP

Alexandru Burlacu

Spring 2020

Actor Model - Details

The fundamental idea of the actor model is to use actors as concurrent primitives that can act upon receiving messages in different ways:

1. Create more actors
2. Send messages to other actors
3. Designate what to do with the next message by changing its own internal behavior

Actor Model - Details

- Actor Model uses asynchronous message passing, i.e. an actor doesn't wait for an acknowledgement of the message by the receiver actor.

- Actor Model does not use any intermediate entities such as channels.
  Each actor possesses a mailbox and can be addressed.

- Addresses != identities.
  Each actor can have no, one or multiple addresses.

- An actor must know the address of the recipient.
  Actors are allowed to send messages to themselves.

Actor Model - Details

Actors Model based systems must be designed with a special property in mind - **Inconsistency Robustness**, i.e. a system must be able to function properly even if the internal state is inconsistent.

Sort of like social systems, like organizations, governments or communities. They are all capable to cope with some degree of inconsistency.

This property is necessary due to the lack of in-order delivery of messages, in the formal description of the Actor Model.

Actor Model - Modern capabilities

Today's implementations of Actor Model, be it embedded in language, like Erlang and Elixir, or available as libraries, like Akka for JVM systems and Celluloid for Ruby, must meet the following capabilities:

- Fault tolerance
- Distribution transparency
- Scalability (local and nonlocal)

Actor Model - Common Patterns

Your default case must be one or more actors per, connection/client/whatever. Actors are light, so it’s not a problem having plenty of them.

Due to their internal rules (being sequential) using one actor for multiple tasks doesn’t let you have maximum concurrency in the system.

Actor Model - Common Patterns

Supervision Tree - Actors come in systems, and their preferred system organization is a hierarchy.

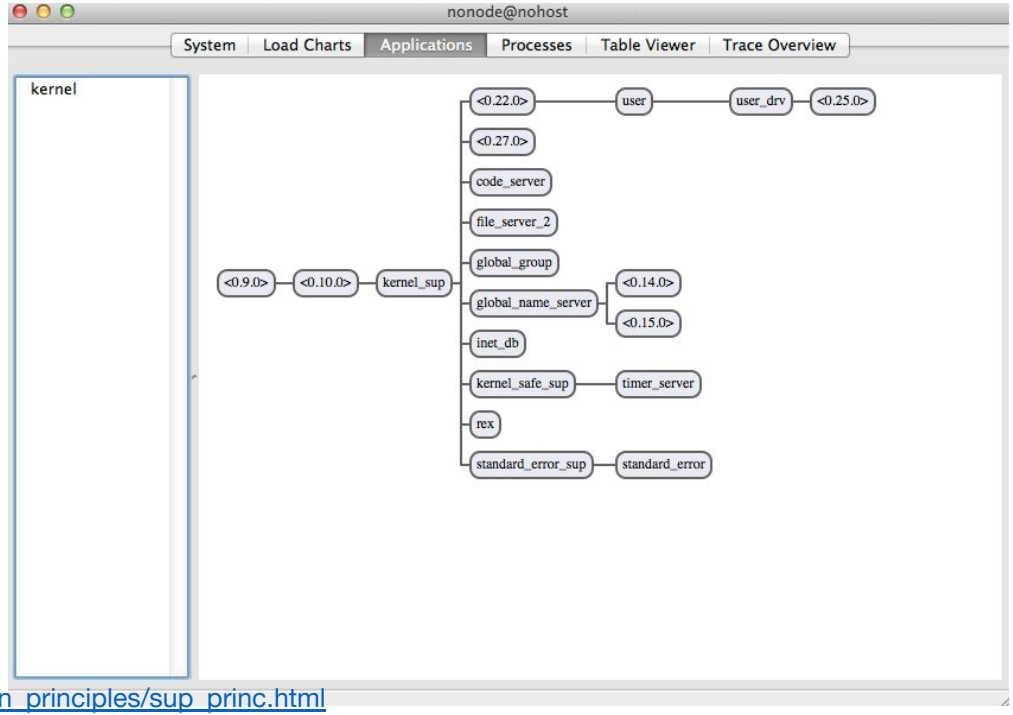
Systems built using the Actor Model tend to be hierarchical, employing special supervisor actors that take care of the worker actors, especially when a failure occurs.

Having supervisor actors taking care of workers makes the overall system robust to failure, a philosophy embraced and popularized by Erlang developers: ***Let it crash***

Actor Model - Common Patterns

Traditionally, there are 2 more popular and one not so restart policy in case of a failed actor:

1. **one-for-one** - restart only the failed actor, for independent tasks
2. **all-for-one** - restart everyone under the supervision, for interacting tasks
3. **(sometimes) rest-for-one** - restart only the actors launched after the failed one



See also: http://erlang.org/documentation/doc-4.9.1/doc/design_principles/sup_princ.html

Actor Model - Common Patterns

Why **Let it Crash** even works?

Well, because eventually something will go wrong, and trying to cover all bad cases is not viable, even your hardware could cause a crash, how will you try-catch it?

Of course, you should aim at writing good code, but sometimes there are errors that aren't even your fault, like hardware, time, or network issues. The kind of errors that occur in the most bizare ways, these are **transient errors**, and Let it Crash/Fail-fast philosophy is something that can save you even in such situations.

Actor Model - Common Patterns

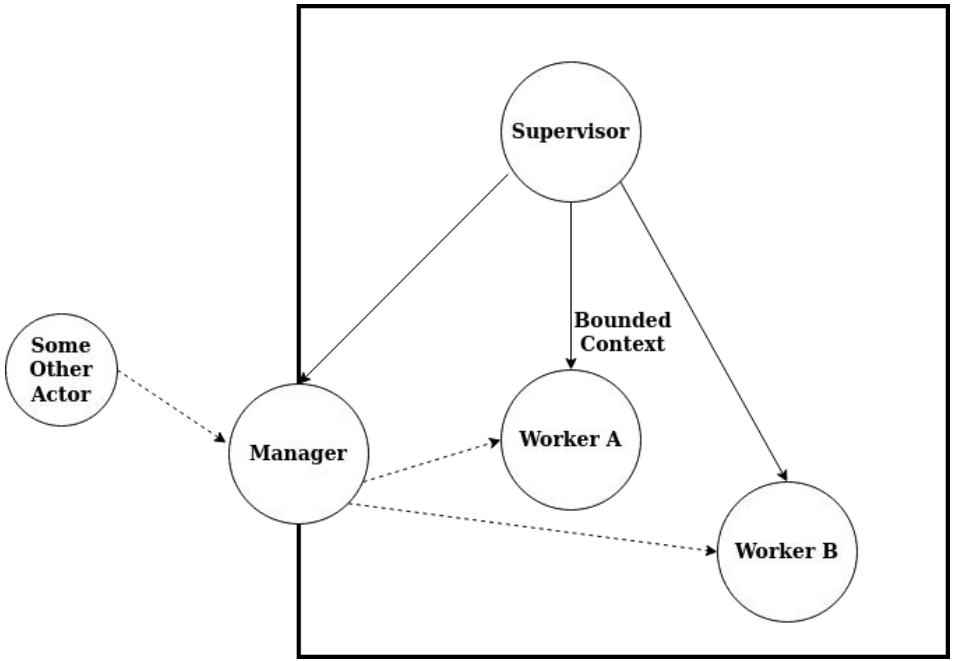
Let it Crash is said to be the philosophy to program the “happy path” and not bother about errors. It’s not entirely like this, more like, if you don’t know how to properly (based on specs) handle the error, don’t bother.

In combination with supervision trees and a couple of other patterns it is possible to achieve impressive system resilience, only because the failure can be localized and handled clearly.

Actor Model - Common Patterns

A way to structure an actor-based system that is using supervision trees, that comes from Erlang community, is to have a **manager** (think of it as a façade pattern) that redirects messages to different **workers** (think controllers).

All of them under a single supervisor with a all-for-one restart policy.



Actor Model - Common Patterns

Another very common pattern when using actor model is having having a publisher-subscriber aka observer pattern implementation. In fact, PubSub is fundamental for most modern reactive systems, but about this later.

This way, it is possible to have topics and entities interested in receiving updates about these topics. Actor model is especially good at it due to its distribution transparency property, and lightweight nature.

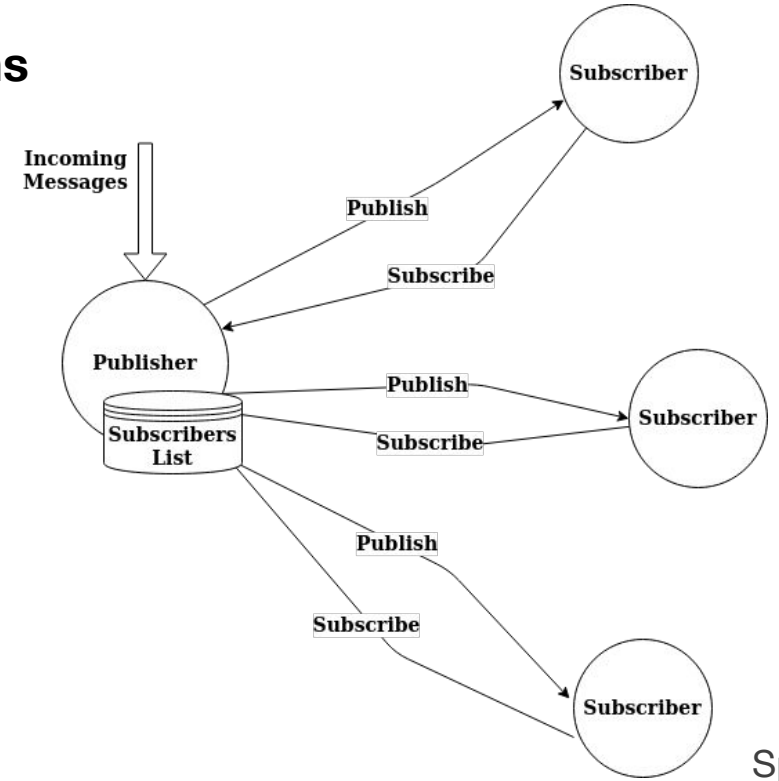
PubSub in such languages as Elixir or Erlang is commonly done via **:gen_event** OTP behaviour, but that's just technicalities.

Actor Model - Common Patterns

A quick reminder of what PubSub is:

You have a set of message/event creators, let's call them **publishers**, and another set of entities that are interested in some, or all of the messages/events, these are **subscribers**.

Subscribers can subscribe to publishers anytime they want and receive updates as soon as possible.



Actor Model - Common Patterns

Character Actor, related in Erlang folklore as the **Error Kernel pattern**, is a way to ensure reliability in case that the system was asked to perform a task with high error chance.

The idea is to create a dedicated actor to perform a task, such as if it fails, it won't damage the rest of the system, while if task completed successfully, the actor is disposed.

During the design process, identify the components that must always be correct, they are the kernel, as in OS parlance, and the ones that could in principle be faulty. This is Error Kernel pattern. Also, in practice, the kernel has a way to write into memory information about its state and the state of its children, for reliability purposes.

Actor Model - Common Patterns

Actor Recursion - for an actor to send a message, it need to know only the address of the recipient. Because of this, an actor can in fact send a message to itself.

This way it could be possible to postpone messages that have lesser priority than others, for example.

Note, actor recursion should not be confused with recursively spawning more and more actors.

Actor Model - Common Patterns

Request-Reply - recall that Actor Model is asynchronous in nature, yet sometimes a way to receive a response is necessary.

How do you imagine that is possible to achieve?

Actor Model - Common Patterns

Request-Reply - recall that Actor Model is asynchronous in nature, yet sometimes a way to receive a response is necessary.

How do you imagine that is possible to achieve?

Add the sender's address/reference to the message sent, and have a way to receive the reply on the server.

Beware, that way you introduce coupling in the system. At least make sure the sender doesn't block while waiting for a reply.

Actor Model - Common Patterns

How do you perform **transactions** within a system using actor model?

Recall what a transaction is - a sequence of operations that either are all successful or discarded, and the state of the system rollbacked to the state before the transaction.

Actor Model - Common Patterns

This is actually a hard problem, and one of the reasons why transactions are rare in actor-based systems.

Generally, if you need transactions, either try using an external database-like component, or design a coordination protocol using messages.

Or even better, rethink your system.

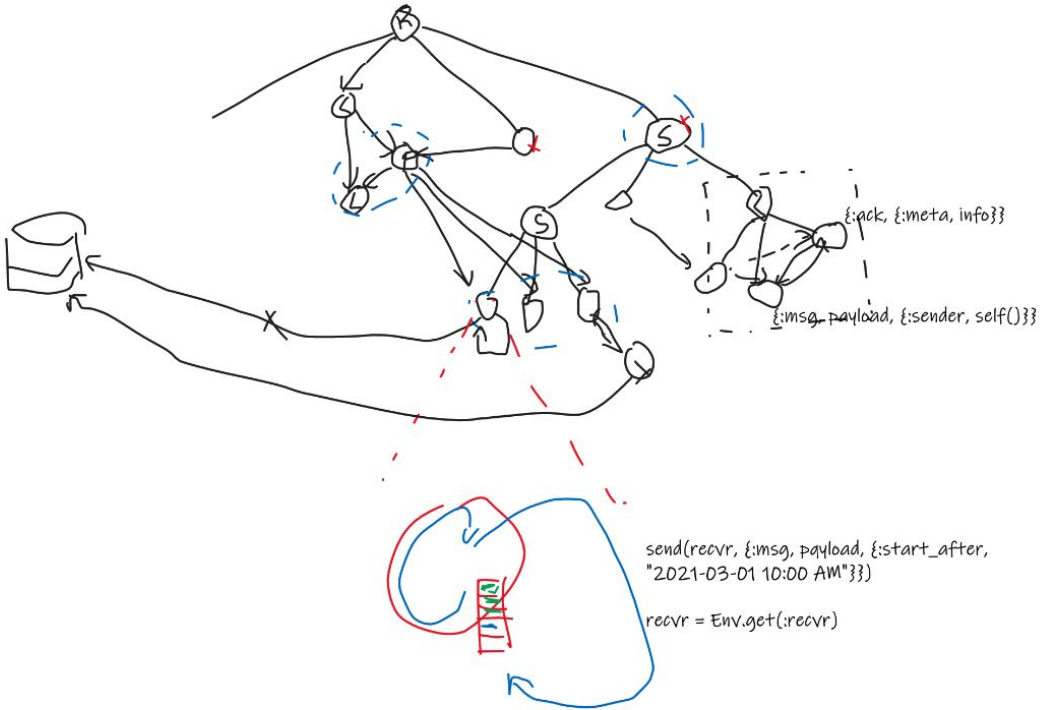
Actor Model - Common Patterns

In Erlang-land there are no definite patterns of doing transactions because of the availability of other tools, namely ETS tables and the Mnesia database.

There was a special class in Akka 2.0.x for JVM called Transactor, today it is deprecated due to Akka's orientation towards distribution transparency.

Anyway, Transactor was capable to enable transaction within an actor, using STM semantics.

Actor Model - Common Patterns Combined



Aka
#SpotThePattern

Actor Model - When not to use it?

On a single laptop one can spawn thousands, if not hundred of thousands of actors. But keep in mind that that same laptop won't have more than 4 physical/8 logical cores, as of 2020.

This discrepancy between physical parallelism and actor model capabilities results in terrible parallel performance. Recall - **Concurrency is not Parallelism!!!**

Therefore, a system using actors, generally, won't work well in CPU bound tasks, like linear algebra routines, or signal processing. On the other hand it is proven that actor model is probably ideal for simulations, due to it's easy mapping to multi agent systems.

Actor Model - When not to use it?

Another more subtle problems with the actor model are **not really composable** and they **couple concurrency and mutual exclusion**, leaving synchronization to be implemented via custom messaging protocols.

1st, actors are not composable because, unless the developer before you was caring, the receiver of a message is hardcoded, therefore, having 2 actors A and B, and A sends messages to C, therefore making it non-trivial to make A send messages to B.

2nd, coupling happens as a result of actors encapsulating pices of state and having internal sequential execution, while due to asynchronous communication, the whole system is concurrent.

See: is broken https://noelwelsh.com/posts/2013-03-04-why-i-dont-like-akka-actors.html

Actor Model - When not to use it?

If you have some state, multiple threads reading and writing from/to that value will end up inducing the possibility of deadlocks and race conditions, beside increased complexity due to synchronization. If you put that state inside an actor, suddenly you can be sure that it's accessed safely and you aren't dropping writes or getting stale reads.

Therefore, as a rule of thumb, use actors when you need safe state.

In case of not Erlang-based languages, Futures and Tasks are simpler and more suitable if you don't need to mutate state.

~~Actor Model~~ A quick not-so-off-topic

Right now we are in the realm of massive concurrency.

Actor model, and further communicating sequential processes, due to their lightweight nature allow anyone to use hundreds, thousands, even millions of concurrently running operations.

Because of this, understanding the limits of system scalability must be understood.

In this section, several formulas will be discussed:

- Little's Law
- Amdahl's Law
- Universal Scalability Law

~~Actor Model~~ A quick not-so-off-topic

Little's Law is a simple equation of the behavior of queues.

It formally describes the relationship of queue size (**N**), throughput (**X**) and latency (**R**):

- **N=XR**
- **X=N/R**
- **R=N/X**

Although not all systems use queues, Little's law still holds. For example when trying to keep the latency at bay when the number of connections to the system surges.

See: https://codahale.com/usl4j-and-you/

~~Actor Model~~ A quick not-so-off-topic

Amdahl's Law - suppose we have a data processing pipeline, processing 10k data points per second, on a 8 core server.

Now, if you will change the CPU to a 16 core one, you will notice that the number of data points processed isn't 20k, more like 17-19k.

Why is that?

~~Actor Model~~ A quick not-so-off-topic

There are portions of the workload that can't run in parallel, called serial workload.
Knowing what the fraction of the workload is serial helps understand how will it scale given more cores/machines/compute units to run on.

$$S(s) = 1/(1 - p + p/s)$$

- **S(s)** is the overall system speedup given the speedup of a certain component in the system.
- **p** is the ratio of the program that is subject to speedup
- and **s** is the speedup of that component

~~Actor Model~~ A quick not-so-off-topic

Universal Scalability Law… Before we begin, here's a question.

Assuming an 8 core CPU on which a program that **maps** a numerical function to a set of values and then **aggregates** the newly formed set of results, again, numerically speaking; what configuration of the worker pool will be faster, 8 workers or 24, or even 64?

~~Actor Model~~ A quick not-so-off-topic

Universal Scalability Law was proposed by N. J. Gunther, as model which combines Amdahl's Law and Gustafson's Law.

It takes into account the **cost of communication** between processes, to produce a nonlinear model to predict a system's behavior when scale changes.

~~Actor Model~~ A quick not-so-off-topic

$$X(N) = \frac{\lambda N}{1 + \sigma(N-1) + \kappa N(N-1)}$$

X is the throughput of the system as a function of the level of concurrency **N (think # of threads or connections).**
Sigma is the overhead of contention, how many contend for a resource
kappa - the overhead of crosstalk, how much time until consistent
and **lambda** is the maximum performance of the system, sometimes omitted.

See: https://codahale.com/usl4j-and-you/
and https://blog.acolyer.org/2015/04/29/applying-the-universal-scalability-law-to-organisations/

~~Actor Model~~ A quick not-so-off-topic

In the end, **Universal Scalability Law** states, and reasonably so, that as the number cores/machines computing something in a parallel/concurrent fashion grows, at some point not only the system will scale slower, but in fact the performance will degrade.

In a way, even context switches can be modeled as communication, due to the cache being flushed for the other thread/worker to be resumed.

~~Actor Model~~ A quick not-so-off-topic

Assuming we scaled our program/system from 1 core to 16, and observed a 10x speedup, we decide to try doubling the computational resources again, and notice only a 12x speedup, compared to the baseline. **We reached the scalability limit for our application**. What next?

More often than not, we are not parallelizing all the parallelizable parts of our programs. Let that sink in. We can have algorithms or systems that 90% of which can be ran in parallel, but when we decide do add more cores into it, we only do so for a part of these 90%.

Identifying where should we optimize next, the bottleneck of our system, is what profiling is used for.

~~Actor Model~~ A quick not-so-off-topic

When **profiling** some code, we measure the time it takes for each function to be executed, how many times it happens, and we do so recursively, going in depth of our program.

Some profilers also allow for memory usage analysis. Nethertheless, using a profiler is the go-to strategy when it comes to optimizing the code, whenever by scaling parts of it, or through other means.

Never, ever try to optimize stuff that isn't confirmed to be the bottleneck of the system. Neglecting this advice can lead to anything from no speedup to an observable slowdown and inefficient resource utilization.

Actor Model - Quiz Time!!

Close your laptops; Get a sheet of paper; For forests worldwide sake, cut it up to A6 size, you won't need much; Share with your colleagues.

Don't worry, the results won't affect your midterm grades. Or maybe they will… or… we shall see.

Actor Model - Quiz Time!!

Q1. Actor Model is based on:

a) Shared state b) Message Passing c) Voodoo magic

Q2. Actor Model's fundamental unit/units of computation is:

a) An actor b) A routine and a channel c) An actor and a mailbox

Q3. A popular pattern for Actor Model systems is:

a) Mediator b) Supervisor c) Adapter

Q4. Actor Model should not be used when doing:

a) CPU-bound tasks b) Transactions c) IO-bound tasks

Actor Model - Quiz Time!!

Q5. The limiting factor when running parallel/concurrent tasks is (assuming ∞ memory):
a) Communication overhead b) Context switches c) Number of threads

Q6. Any system can be modeled good enough, throughput/latency-wise:
a) Stack b) Queue c) Black Box

Q7. Erlang is fault tolerant because of:
a) Actors b) Supervisors c) it being Swedish-born

Q8. “Keeping the critical logic inside a root-actor” pattern is called:
a) Error Kernel b) Transactor c) Character Actor

Actor Model - Some links

- https://ferd.ca/the-zen-of-erlang.html
- https://theory.stanford.edu/~jcm/cs358-96/actors.html
- http://ulf.wiger.net/weblog/2008/02/06/what-is-erlang-style-concurrency/
- https://stackoverflow.com/questions/8107612/the-actor-model-why-is-erlang-special-or-why-do-you-need-another-language-for
- http://www.perfdynamics.com/Manifesto/USLscalability.html

CSP

Communicating Sequential Processes

Alexandru Burlacu

Spring 2020

CSP - Some History

CSP, aka Communicating Sequential Processes is another well known message passing approach to concurrency. Same as Actor Model, only different.

CSP was first proposed by Tony Hoare in 1978, as a way to formally specify the behaviour and model concurrent systems.

Initially it was closer to Hewitt's Actor Model, also requiring the address of the recipient to be known, but later it switched to channels and anonymous processes.

CSP - Theory 101

CSP has 2 main primitives, **events** and **processes**. They can be combined using different operators.

Events are fundamental for CSP. They are values that determine the behaviour of the system.

Processes are anonymous in nature and either process or react depending on the event.

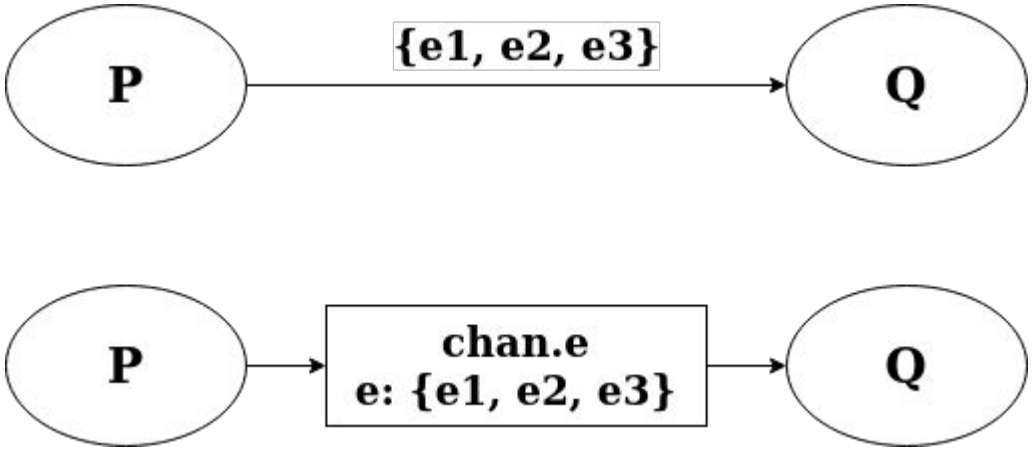
What about **channels**?

CSP - Theory 101

Channels, which are used in practice are not really defined in CSP formalism. Rather, these are related to the events.

Say, you can have an ***inp*** and ***out*** channel to your process but what matters to the underlying theory are the ***inp.x*** and ***out.y*** events, where ***x*** and ***y*** are values from the ***inp*** and ***out*** channels respectively.

CSP - Theory 101



CSP - Theory 102

Some of the operators specified by the CSP formalism are:

- **choice** operators, deterministic and non-deterministic
- **interleaving**, that is running 2 independent processes concurrently
- **prefixing**, an event and a process, together specifying a new process
- **interface parallel**, that is running 2 processes concurrently that need synchronization at some point. Synchronization is done via both processes being able to process an event before that event can occur.
- **hiding**, allows to simply CSP expressions, sometimes + can abstract some processes away
- **sequential composition**, given processes P and Q, if P runs successfully, run Q after that

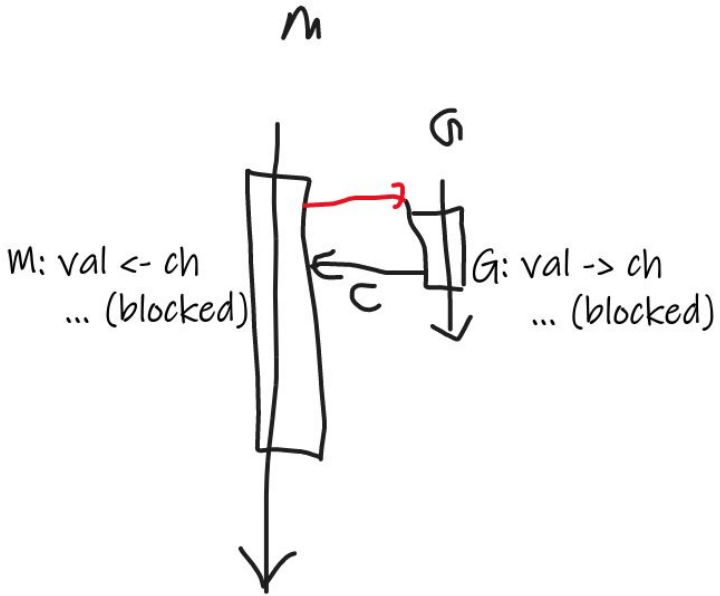
CSP - Theory 102: Syntax

$Proc$	$::=$	STOP	
	$\mid$	SKIP	
	$\mid$	$e \rightarrow Proc$	(prefixing)
	$\mid$	$Proc \,\Box\, Proc$	(external choice)
	$\mid$	$Proc \sqcap Proc$	(nondeterministic choice)
	$\mid$	$Proc \mid\mid\mid Proc$	(interleaving)
	$\mid$	$Proc \mid[\{X\}]\mid Proc$	(interface parallel)
	$\mid$	$Proc \setminus X$	(hiding)
	$\mid$	$Proc; Proc$	(sequential composition)
	$\mid$	if b then $Proc$ else $Proc$	(boolean conditional)
	$\mid$	$Proc \triangleright Proc$	(timeout)
	$\mid$	$Proc \,\triangle\, Proc$	(interrupt)

Source: https://en.wikipedia.org/wiki/Communicating_sequential_processes for full list check CSP Book

CSP - Theory 102: But how is it concurrent if it's synchronous?

After synchronization, 2 processes can run independently, that's how.



M: e1 -> e2 -> ch.val -> (1 -> M;
2 -> e3 -> M)

G: eg1 -> eg2 -> ch.in -> G

Actors vs CSP

Actor Model	Communicating Sequential Processes
Same: - Message oriented - Used for modeling/formal specification	Same: - Message oriented - Used for modeling/formal specification
Different: - Processes/Actors have identity - Asynchronous by default - Built-in unbounded nondeterminism	Different: - Anonymous processes - Synchronous by default - Only bounded nondeterminism

Actors vs CSP

Practically equivalent. One can build CSP using Actor Model and vice versa

Alexandru Burlacu

Spring 2020

CSP - Common Patterns

“With great power comes great responsibility” - Uncle Ben minutes before dying, telling Peter Parker not to abuse CSP in projects.

Main things to keep in mind when considering using CSP-like abstractions in your project are:

1) How expensive is it to work with, system resources-wise.
   - Is it cheap to have many processes?
   - Is it cheap to switch between them?
   - Where in the codebase is it critical to have low latency access to resources? **Hint**: Look for contentions.
2) Is it the right abstraction to work with? **Hint**: Might not be.

CSP - Common Patterns

Before we start, the examples will be in a pseudocode similar to the Go programming language.

```
ch <- val // put a value `val` into the channel `ch`
val <- ch // assign to value `val` what you get from channel `ch`
```

`for` Loops and `if` statements have no `()`.
`func name?(in_name: in_type): out_type` defines a function, with optional name.
`go` launches a concurrent process.

For example:

```
go func basic(inp: ch, out: ch): void { v <- inp; out <- process(v); }
```

CSP - Common Patterns

Futures are a CS concept that describes a proxy object for a result that is initially unknown, usually because the computation of its value is not yet complete.

Depending on the language/framework in use, they can be sometimes called tasks, promises, delays or deferred.

In scenarios where you know beforehand that you will need a value, you can start computing it on another processor and have it ready when you need it.

Futures run just once.

CSP - Common Patterns

```
func Future(arg: ArgType): ResTypeChannel {
    ResTypeChannel future;
    go func () { future <- Operation(arg)  }();
    return future;
}

// And the client code...

future_result = Future(some_arg);
result <- future_result;
```

CSP - Common Patterns

Generators are a concept that describes an concept that looks like a function, i. e. can have parameters and returns a sequence of values (in the end), but in fact, it is an iterator, that is, return values one by one, and abstract away how these values are stored.

Generators are a weaker form of coroutines. They too can suspend their execution, but can not specify to whom yield the execution context.

Generators can run more than just once. Potentially these can run indefinitely.

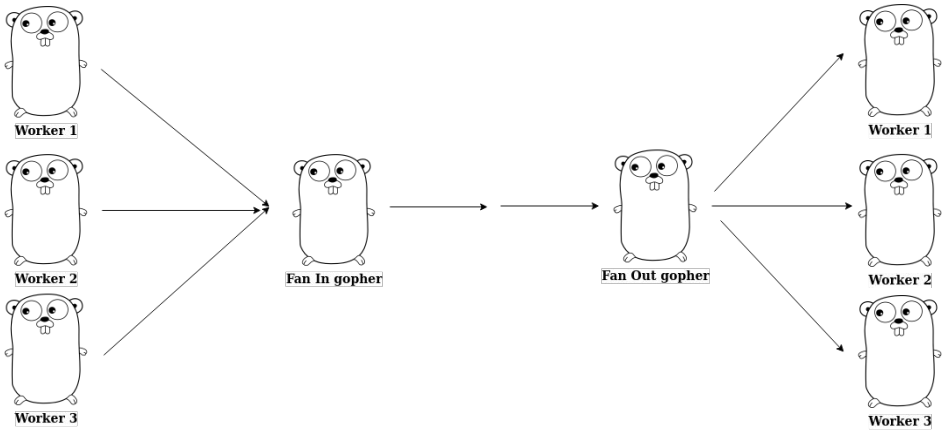
CSP - Common Patterns

```
func GeneratorAdInfinitum(): ResTypeChannel {
    ResTypeChannel gen;
    go func () {
        for i := 0; ; i++ {
            gen <- i;
        }
    }(); return gen; } // The client code below

gen = GeneratorAdInfinitum();
next <- gen; // 1
next <- gen; // 2
next <- gen; // 3
```

CSP - Common Patterns

Say you have many concurrently running processes and you want to have a way to collect data from all of them. Or just the opposite, distribute work among processes. Then you need Fan-In and Fan-Out respectively.



CSP - Common Patterns

```
func FanInProc(...chs: Channels): ResTypeChannel {
    ResTypeChannel fan_in;
    for ch in chs {
        go func () { for { fan_in <- <- ch; } }();
    }
    return fan_in;
} // The client code below

gen = FanInProc(ch1, ch2, ch3);
next <- gen; // ch1_data
next <- gen; // ch3_data
next <- gen; // ch2_data
```

CSP - Common Patterns

```
func FanOutProc(ch_in: Channel, chs_out: ListOfChannels): void {
    for idx, val in enum(ch_in) {
        go func () { chs_out[idx % len(chs_out)] <- val; }();
    }
} // The client code below

FanOutProc(producer_ch, [consumer_ch1, consumer_ch2]);
// prod_data_1 -> consumer_ch1
// prod_data_2 -> consumer_ch2
// prod_data_3 -> consumer_ch1
// ...
```

CSP - Common Patterns

Semaphores can be easily implemented using buffered channels. Although CSP focuses on unbuffered channels for brevity, in practice it is possible to add a size to the channel, thus making the communication asynchronous.
In such case, a semaphore is a buffered channel of some tokens.

```
// acquire n resources
func (s semaphore) P(n int) {
    e := empty{}
    for i := 0; i < n; i++ {
        s <- e
    }
}
```

```
// release n resources
func (s semaphore) V(n int) {
    for i := 0; i < n; i++ {
        <-s
    }
}
```

CSP - Common Patterns

Having semaphores, it is now possible to implement mutexes and signaling mechanisms.

```
// mutex
func (s semaphore) Lock() {
    s.P(1)
}

func (s semaphore) Unlock() {
    s.V(1)
}
```

```
// signaling
func (s semaphore) Signal() {
    s.V(1)
}

func (s semaphore) Wait(n int) {
    s.P(n)
}
```

CSP - Common Patterns

Sometimes, there's a need to call multiple functions but only a single result will suffice.

Say, you want to query a set of databases and need to return the first arrived message.

In a language like go, you could do something like the snippet on the right.

```
func Query(
    conns []Conn,
    query string) Result {
    ch := make(chan Result)
    for _, conn := range conns {
    go func(c Conn) {
        select {
        case ch <- c.DoQuery(query):
        default:
        }
    }(conn)
    }
    return <-ch
}
```

CSP - Common Patterns

Regretfully the previous pattern results in K times more queries, where K is the number of connections.

We could modify the code on the left to query the next connection after some time, rather than right away, thus substantially reducing the load on the servers.

This is called **speculative execution**, or sometimes **hedged requests**.*

* https://pdos.csail.mit.edu/6.824/papers/tail-dean.pdf

```
func QueryV2(conns []Conn,
    query string,
    time_out time.After) Result {
    ch := make(chan Result)
    var query func([]Conn)
    query = func(cs []Conn) {
        select {
        case ch <- c.DoQuery(query):
        case <- time.After(time_out):
          go query(conns[1:])
        }
    }
    go query (conns)
    return <-ch
}
```

CSP - Common Patterns

A lot of good stuff is done using **pipelines**.

```
cat urls.txt | xargs -I % curl % |
echo - >> responses.txt
```

From the point of view of CSP, the `|` (pipe) is a channel and the code between the pipes - processes.

```
func Proc(
    in <-chan string) <-chan string {
    ch := make(chan string)
    for url := range <-in {
        go func(u string) {
            makeRequest(u)
        }(url)
    }
    return <-ch
}
```

CSP - Common Patterns

A lot of good stuff is done using **pipelines**.

```
cat urls.txt | xargs -I % curl % |
echo - >> responses.txt
```

From the point of view of CSP, the `|` (pipe) is a channel and the code between the pipes - processes.

Pipelines are good because each step can run at the same time.

```
func Proc(
    in <-chan string) <-chan string {
    ch := make(chan string)
    for url := range <-in {
        go func(u string) {
            makeRequest(u)
        }(url)
    }
    return <-ch
}
```

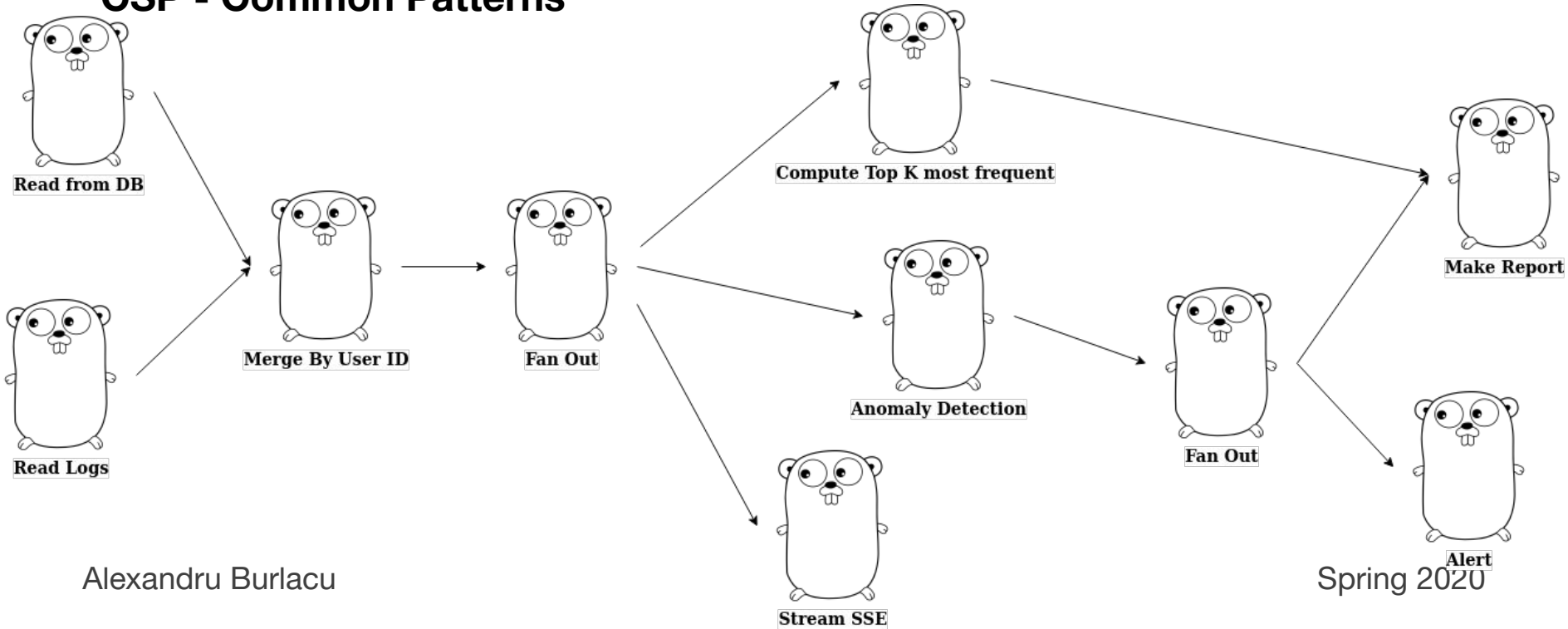
CSP - Common Patterns

Combined with **fan in** and **fan out** patterns, it is possible now to create directed acyclic graphs of computations. You should be excited now!

DAGs are a natural fit for data processing use cases. Many modern tools rely on this abstraction.

In fact, it is possible to define cyclic graphs too, but better don’t.
Cyclic graphs would result in much more complicated logic and debugging.

CSP - Common Patterns



CSP - Channel Size

By default, channels in CSP are single element, or rather, are synchronization variables.

So, if I have processes **P** and **Q** and a common channel/event **ch** - **(ch!in -> P) || (ch?out -> Q)** means that if I put a value in **ch**, my process will block until that value is fetched. This gives easy way to synchronize processes, but is not always desired.

In practice, one can have buffered channels, that for example help a tiny bit when the producing process **sometimes** works faster than the consuming one.

If that happens more than just sometimes, than different flow control techniques must be used.

~~CSP~~ A quick not-so-off-topic #2

Now you already know pretty well both CSP and Actor Model. At least I hope so.

You know their main features, when to use them, and how to reason about these systems.

It's probably time to learn how to **divide** work between **concurrent (or parallel)** entities to achieve maximum performance.

~~CSP~~ A quick not-so-off-topic #2

When it comes to parallel and concurrent program design, it’s mostly about decomposition.

There are two main points of view that you must think of in the process.

Data and computation.

~~CSP~~ A quick not-so-off-topic #2

To achieve maximum performance, you must take into account the system your programs will be running on.

Taking hardware into consideration when designing software is sometimes known as **Mechanical Sympathy**.

A term first used not in software engineering, but more on that later. Like, a couple of weeks later.

Now let's focus on pure (almost) software.

~~CSP~~ A quick not-so-off-topic #2

To design an efficient program running on a parallel processor and/or using concurrency efficiently we need to understand the **problem** and what are the general **patterns** of solving it.

As said earlier, programs can be considered from the point of view of data, and of computation.

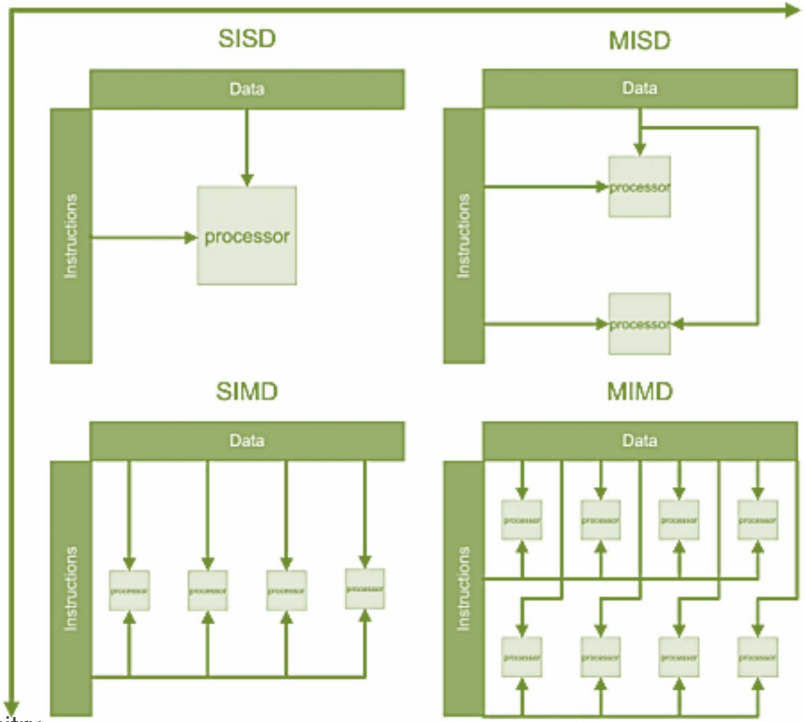
Using Flynn's Taxonomy we can see the relations between different data-computation partitions.

~~CSP~~ A quick not-so-off-topic #2

This is Flynn's Taxonomy.

SISD - single instruction, single data
SIMD - single instruction, multiple data
MISD - multiple instruction, single data
MIMD - multiple instruction, multiple data

Also consider:
SPMD and MPMD, where instead of Instructions (I), Programs (P) are considered, relaxing the **lockstep** requirement.



Source: PAD course, "Distributia: spatii de decentralizare", conf. univ. Ciorba Dumitru

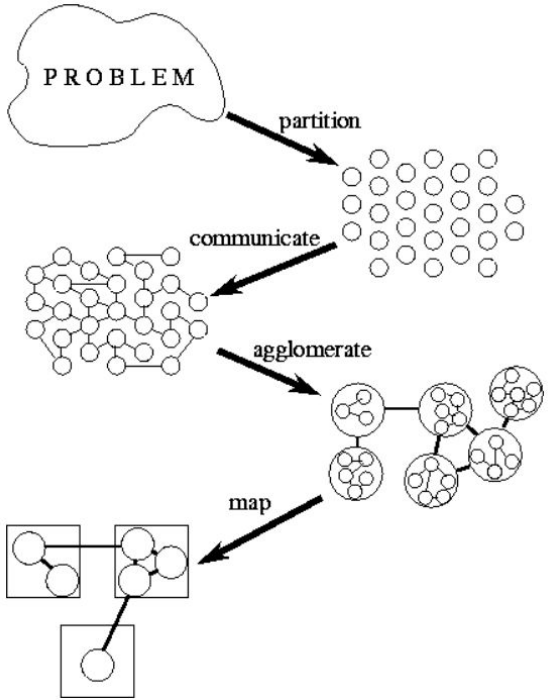
~~CSP~~ A quick not-so-off-topic #2

The objectives of parallel program design are:

- Maximize resource utilization
- Minimize communication overhead
- Ensure equal work distribution (Load balancing)

There are four steps in this process, as of Ian Foster:

- Decomposition
- Communication
- Agglomeration
- Mapping



Source: “Designing and Building Parallel Programs”, Ian Foster

~~CSP~~ A quick not-so-off-topic #2

Recall from earlier, the pipeline pattern from CSP.

This is the most basic way to achieve **task parallelism**. Task parallelism means dividing the program into distinct tasks, each running independently and communicating with others.

Maybe you can even recall the example of parallel map over a sequence of values, from last course (Network Programming). Well, that one is the most basic example of **data parallelism**.

~~CSP~~ A quick not-so-off-topic #2

Now, recall the DAG, also from the CSP lectures earlier.

DAGs can be used to achieve both Data and Task parallelism.

In fact, a DAG structured program can model any of the 4 variants of Flynn's Taxonomy… almost, if we ignore the lockstep requirement.

Enough theory, let's analyze an example!

~~CSP~~ A quick not-so-off-topic #2

First, consider a naive matrix multiplication algorithm:

```
for i in 0..M:
 for j in 0..N:
   for k in 0..K:
     C[i, j] += A[i, k] * B[k, j]
```

How would you parallelize it?

~~CSP~~ A quick not-so-off-topic #2

```
for i in 0..M:
 for j in 0..N:
   for k in 0..K:
     C[i, j] += A[i, k] * B[k, j]
```

My proposition would be to make the outermost loop parallel. Why?

```
parallel for i in 0..M:
 for j in 0..N:
   for k in 0..K:
     C[i, j] += A[i, k] * B[k, j]
```

~~CSP~~ A quick not-so-off-topic #2

```
parallel for i in 0..M:
 for j in 0..N:
   for k in 0..K:
     C[i, j] += A[i, k] * B[k, j]
```

Modern processors have special instructions allowing SIMD operations. GCC and JVM (to a lesser extent) can identify cases where it will work and apply the optimization.

```
parallel for i in 0..M:
 for j in 0..N:
   for k in 0..K..bsize: # bsize - batch size
     C[i, j] += A[i, k:k+bsize] * B[k:k+bsize, j]
     # let's hope, and use -O2 and -mavx2 GCC flags to get more performance
```

~~CSP~~ A quick not-so-off-topic #2

Now, let's say we have a chain of operations where we want to multiply 2 matrices and the result to be added to the 3rd matrix.

The best way to do it, is to have the 2 operations combined, or "fused". Btw, modern CPUs can also combine multiplication and addition into a single very efficient operation.

```
for b in 0..(M/bsize):  # bsize - batch size
 parallel for i in 0..M:
   for j in 0..N:
     for k in 0..K:
       temp = A[i, j, k] * B[i, k, b] + D[i, j, b]
       C[i, j, b] += temp
       # let's hope, and use -O2, -mavx2, AND -mfma GCC flags
```

~~CSP~~ A quick not-so-off-topic #2

We all know there are better algorithms to multiply matrices, like Strassen algorithm.
Recall, it splits a matrix in blocks, then creates intermediary matrices M1 to M7 and multiplies them:

```
M1 = multiply(add(A11, A22), add(B11, B22));
M2 = multiply(add(A21, A22), B11);
M3 = multiply(A11, sub(B12, B22));
M4 = multiply(A22, sub(B21, B11));
M5 = multiply(add(A11, A12), B22);
M6 = multiply(sub(A21, A11), add(B11, B12));
M7 = multiply(sub(A12, A22), add(B21, B22));
```

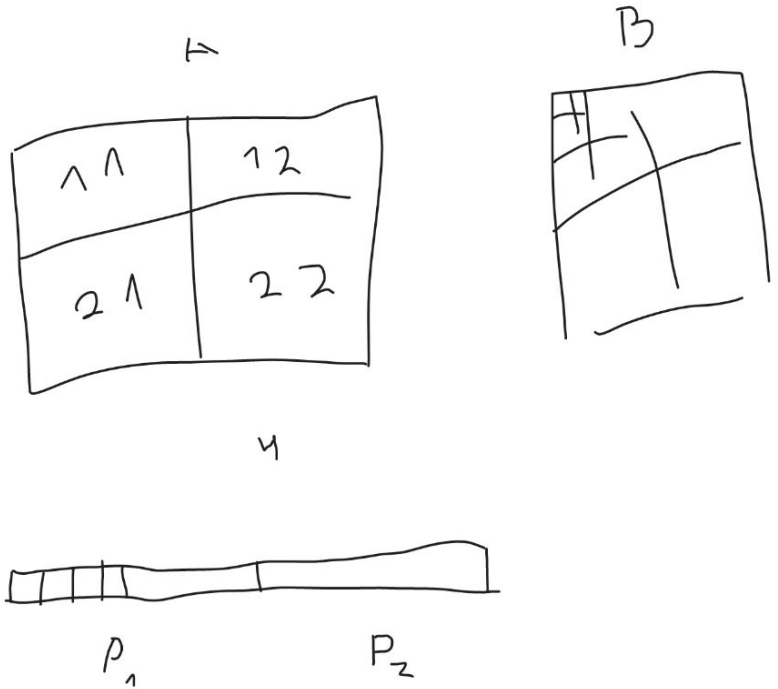
Although not as performant in parallel as naive matmul, Strassen algorithm is a good case of divide et impera (you should know that) which results into a **recursive data decomposition**, in contrast with block decomposition of the naive matmul.

~~CSP~~ A quick not-so-off-topic #2

You can see that the matrix A on the left is decomposed in blocks, while based on that B is decomposed recursively.

Although recursive decomposition is more flexible, it usually requires more coordination.

Some structures, like graphs, and problems are more efficiently parallelized using recursive data decomposition.



Reactive Programming

Alexandru Burlacu

Spring 2020

Reactive Manifesto

Reactive Manifesto was first published in 2013, with an update in 2014 that proposes a new mindset of system design, based on asynchronous **message passing** for communication, allow decoupled entities to scale up and down, or be **elastic**, also, because of the 2 properties, such systems would be **resilient** against failures, allowing components to fail and recover independently, and as a result be **responsive**.

You should really read it: https://www.reactivemanifesto.org/

Reactive vs Event-Driven

Although both are working with events, there's a subtle difference between Reactive programming and Event-Driven programming, which entirely changes the game.

Event-driven programming model treats each event as a separate entity, whereas Reactive programming, and specifically Reactive Streams are about treating events as parts of a stream, and the go-to abstraction becomes the stream.

Reactive Manifesto in the small

Quite around the same time as the Reactive Manifesto was proposed, a set of tools appeared, called ReactiveX and a specification - **Reactive Streams**.

The core idea - treat events as an infinite stream of discrete values and work with these as they arrive. Basically, it's a combination of:

- Observer/PubSub Pattern (being notified of arrival, decoupling the Observers/Consumers)
- Iterator Pattern (obtain the elements of an aggregate object without knowledge of it's representation)

Now, why mix the Reactive Manifesto and Reactive Streams? Because one can treat external services which send async messages as sources of data which can be "observed". So Reactive Streams would be a way to handle asynchronous messages both in big and small.

Reactive Manifesto in the small

One important thing to understand about the inner workings of any Reactive Streams library is that it is “push” based, as opposed to “pull”-based of a classical iterator.

Iterator/pulling data:

```
aList.stream()
     .map({ s -> return s * 3.14})
     .forEach({ println "next => " + it })
```

Observable from Rx/pushing data:

```
getMouseXLocation().debounce(100, TimeUnit.MILLISECONDS)
     .map({ s -> return s / 100. })
     .subscribe({ println "on next => " + it })
```

Simple JS Implementation: https://netbasal.com/javascript-observables-under-the-hood-2423f760584
Complex JS Implementation: https://blog.eyas.sh/2019/07/learning-by-implementing-observables/

Reactive Manifesto in the small

One important thing to understand about the inner workings of any Reactive Streams library is that it is “push” based, as opposed to “pull”-based of a classical iterator.

Iterator is being pulled by the client code via `next()` method, whereas the Reactive Streams observable can be subscribed to, which means that the subscriber/observer has to react on some events, usually by specifying the `onNext(smth)` method.

The observers also specify the `onComplete()` and `onError(err)` methods.

ReactiveX - Why?

Thinking reactive makes event-driven programs easier to work with. Including UIs, event processing and notification systems.

Now, it is possible to define behaviours like “if there were more than 5 clicks within 2 seconds on the button, change its color” using the event-driven paradigm, but it’s significantly more cumbersome.

ReactiveX - How?

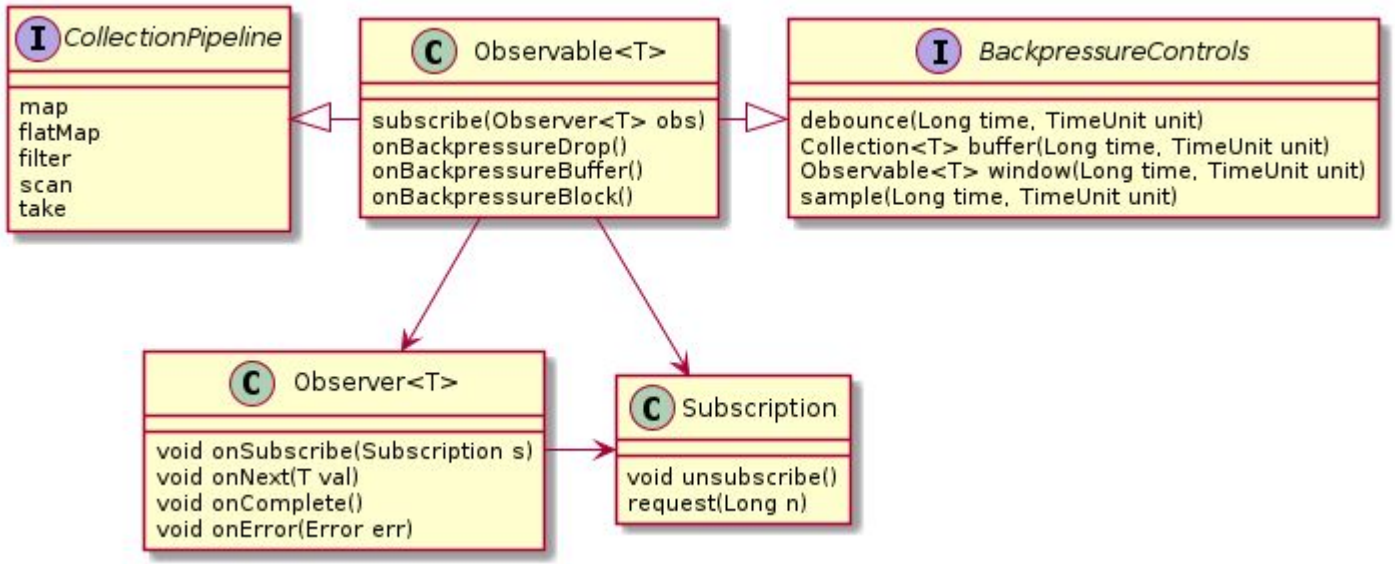
First and foremost, the ReactiveX set of libraries, for many programming languages, including Swift, Java, C++, Python and JS, to mention a few.

Also, the core ideas are available in Akka (remember it?). Even if of higher granularity, the principles are still the same.

Most recently, since Java 9, using Flow API will give you pretty much the same capabilities.

Oh, and since most of you are learning Elixir, check out GenStage ;)

ReactiveX - How?



Inspired by Java 9 Flow API: https://community.oracle.com/docs/DOC-1006738

Backpressure

Recall what congestion control is from the Network Programming course.

Also, recall the relation between the congestion window and the receive window in TCP.

Backpressure

Well, backpressure is related with congestion, only from a different point of view. In practice it focuses on different use cases.

Whereas congestion is usually a phenomenon in the communication channel, be it network or some abstract channel, backpressure is a phenomenon resulting from the consumer who can’t cope with the amount of items arriving.

Backpressure

TCP uses a so-called open loop congestion control mechanism, with its congestion windows.

In principle, backpressure can use a similar method, with minor tinkering with the timing of the ack-s, and spoiler alert: it is used, sort of.

Generally, backpressure is a closed loop mechanism that **notifies** the producer that it can't handle anymore traffic. Depending on the type of the producer, different mechanisms could be used.

Backpressure: Hot & Cold producers

There are 2 commonly known types of producers - hot and cold. The difference is in when and how these are firing events.

Hot producers are the one that abstract away external sources with external control. They will fire events no matter if there are subscribers listening or not.

Cold producers, on the other hand, are the ones that emit events only when there's someone listening. Cold producers can be quite easy handled.

A mouse listener is a hot producer. A file reader is a cold producer. And so is a producer emitting events at specified intervals of time.

https://www.davesexton.com/blog/post/Hot-and-Cold-Observables.aspx

Backpressure: Methods

Three common methods to handle non-blocking backpressure are known. If blocking is ok: block.

- **Sampling** or **dropping** of events sometimes is ok, for example in case of logging/tracing of non-error events.
- **Batching** is another valid scheme. Collect events into groups and emit the groups at specified intervals of time, or number of values in the group. Thus, reduce the rate of fire.
- **Buffering**, or having a queue to moderate the traffic, is only viable if the rate of arriving events/items is variable.
  Do not use buffering when producer constantly outperforms the consumer!

For methods used by RxJava: http://reactivex.io/documentation/operators/backpressure.html

Backpressure: “reactive pull”

Finally, there’s a method called “reactive pull” which basically forces the library user to handle backpressure.

The idea is simple, at the end of the `onNext()` method call the Observer is responsible for asking for more events. It doesn’t mean it’s pulling them. It just asks. Commonly, it asks for one event, but in principle, can ask for more. **Reactive pull is not usable with hot producers.**

```
someObservable.subscribe(new Subscriber<T>() {
    public void onStart() { request(1); }
    // `onComplete` and `onError` omitted for brevity
    public void onNext(T n) {
      // do something with the emitted item "n", then request another item:
      request(1); // if `Long.MAX_VALUE`, not 1, producer will emit at its own rate
    }}); // RxJava 1.x example
```

~~Reactive Programming~~ A quick not-so-off-topic #3

*“You don’t have to be an engineer to be be a racing driver, but you do have to have **Mechanical Sympathy**”* - Jackie Stewart, F1 driver

~~Reactive Programming~~ A quick not-so-off-topic #3

Mechanical Sympathy means that in order to squeeze the most out of your system (application) you must be conscious about its underlinings (hardware).

Good examples of applied mechanical sympathy is designing high performance web servers with thread context switches times in mind.

Or designing C++ numerical libraries, taking into account the cache structure of CPU architectures.

For this off-topic, we will focus more on optimizing memory usage, by applying the idea of **locality**.

~~Reactive Programming~~ A quick not-so-off-topic #3

Locality, as the name implies, is about things that are close to each other. In the context of software performance locality means that necessary things are efficiently accessed. Locality can be spatial and temporal.

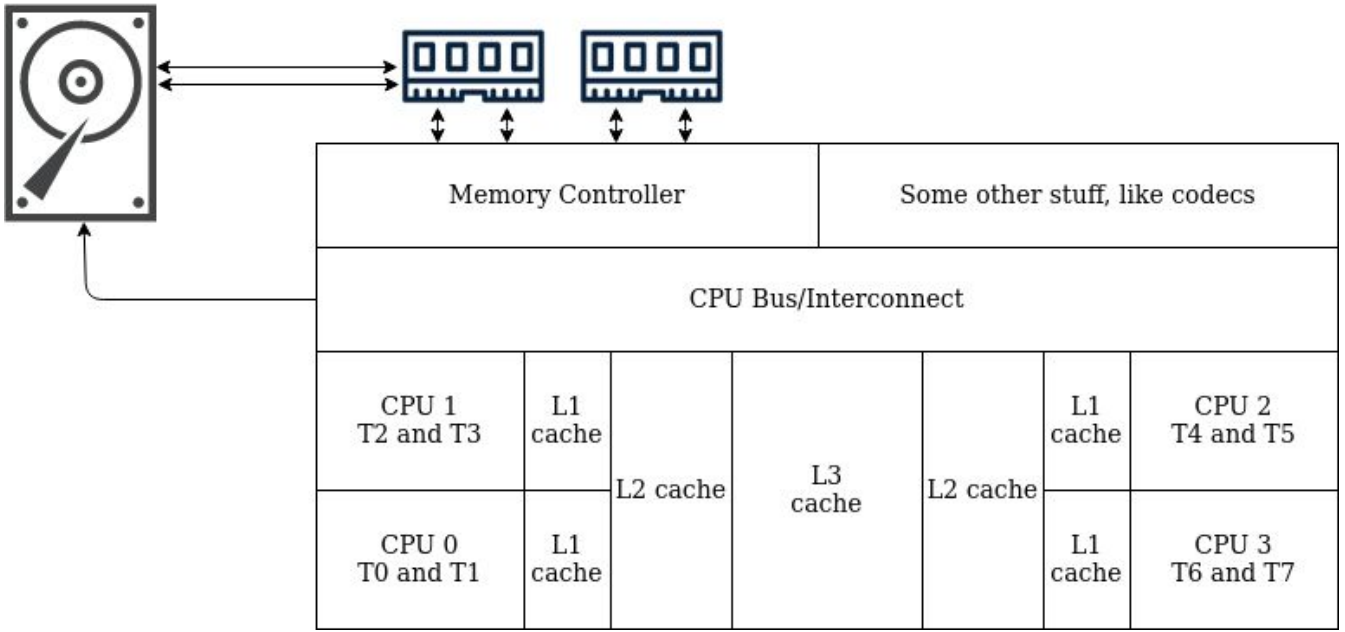
Most of the time we're talking memory or **spatial locality**. Basically, you want to access your memory in a predictable way that minimizes the times you hit the memory, or even the caches. Best way to achieve this is via sequential access for disks and contiguous arrays for things stored in memory.

A bit different, yet still valid, is CPU affinity (btw sometimes called CPU locality). It means you want your communicating threads to be mapped on processors that are closer to each other, to minimize communication cost and context switching times.

~~Reactive Programming~~ A quick not-so-off-topic #3

Short recap of modern CPU architectures and memory hierarchy.

Registers (per CPU)
-> L1 cache
-> L2 cache
-> L3 cache
-> RAM (or simply memory)
-> Disk (SSD sometimes)



~~Reactive Programming~~ A quick not-so-off-topic #3

Based on the idea of CPU affinity, 2 threads that interact the most should be placed either in Tk and Tk+1 where k is odd, if they don't hold a lot of memory, or CPU k and CPU k+1 for k in {0, 2}. The choice is based on the speed of cache access, and therefore its cleaning.

Always remember the stats below.

```
Latency Comparison Numbers (~2012) by Jeff Dean, reduced
----------------------------------------------
L1 cache reference                          0.5       ns
Branch mispredict                           5         ns
L2 cache reference                          7         ns  14x    L1 cache
Main memory reference                       100       ns  20x    L2 cache, 200x L1 cache
Compress 1K bytes with Zippy                3,000     ns  3 us
Send 1K bytes over 1 Gbps network           10,000    ns  10 us
Read 4K randomly from SSD*                  150,000   ns  150 us ~1GB/sec SSD
Read 1 MB sequentially from memory          250,000   ns  250 us
```

~~Reactive Programming~~ A quick not-so-off-topic #3

Temporal Locality is related to structuring the program in such a way that things are accessed in a clustered in time way. In human tongue - you try to re-access things that you recently accessed.

Why? Reducing the time between referencing some memory space/variable/whatever maximizes the chance to get it from the closest (read most performant) cache. For example:

```
int A[10]; A[1]; A[2]; A[4]; A[2]; A[2]; A[4];
```

Although not very useful, this access pattern shows how some values are repeatedly accessed and because of it are likely kept in the closest possible cache to you, thus reducing latency/increasing throughput of operations.

Spatial vs Temporal locality SO: https://stackoverflow.com/a/16554721

~~Reactive Programming~~ A quick not-so-off-topic #3

Cache structure
You might already be familiar with the notion of a cache. Generally it's some sort of key-value store (KVS). Now, CPU caches are similar, not identical to KVS, and are designed to contain **lines of caches**. Think of a cache line as if the value in a KVS is actually a set of values.

There are 3 basic types of caches/cache placements:

1. Direct mapping
2. N-Way (set) associative mapping
3. Fully associative mapping

Understanding how the cache works is fundamental for a software engineer, even if he doesn't aim to squeeze the last effective FLOP/IOP from the hardware.

~~Reactive Programming~~ A quick not-so-off-topic #3

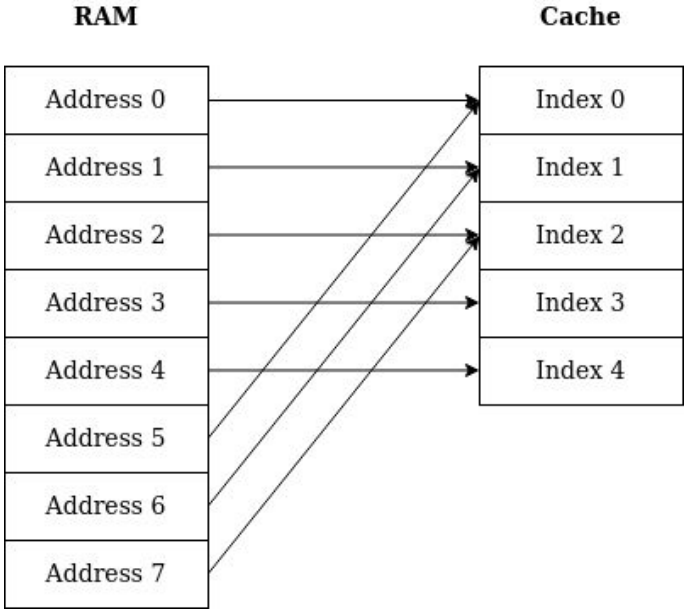
Cache structure: Misc
First thing that comes to mind when talking about caches is how to get rid of surplus data. That is, eviction policy. Some of the simplest, yet universally applicable are: Least Recently Used (LRU), Least Frequently Used (LFU), and FIFO. As the names imply, LRU discards the item that was used least recently, while LFU the one that for some time was referred to least frequently, while FIFO will work as a queue. **Remember, for different use case scenarios other policies might work better.**

Another thing to keep in mind when talking about caches is coherence. Like the movie.
Recall, every CPU has its own L1 and sometimes L2 caches, and it might be the case that multiple CPUs work with the same value. Now, we need a mechanism that will **propagate writes** and **order transactions** to all caches containing the value. Because lacking such mechanism can lead to different values, in different caches, for the same cache entry. For that we have **cache coherence protocols**.

~~Reactive Programming~~ A quick not-so-off-topic #3

Cache structure: Direct mapping
Direct mapping means the memory address from the RAM can be inserted only in a single cache entry. It's fast, but unpredictable, leading to quite bad cache hit ratio, which btw we try to maximize.

Let **x** be block number in cache, **y** be block number of memory, and **n** be number of blocks in cache, then mapping is done with the help of the equation **x = y mod n**. If a collision occurs, than the older value is swapped with the newer one, until next collision.

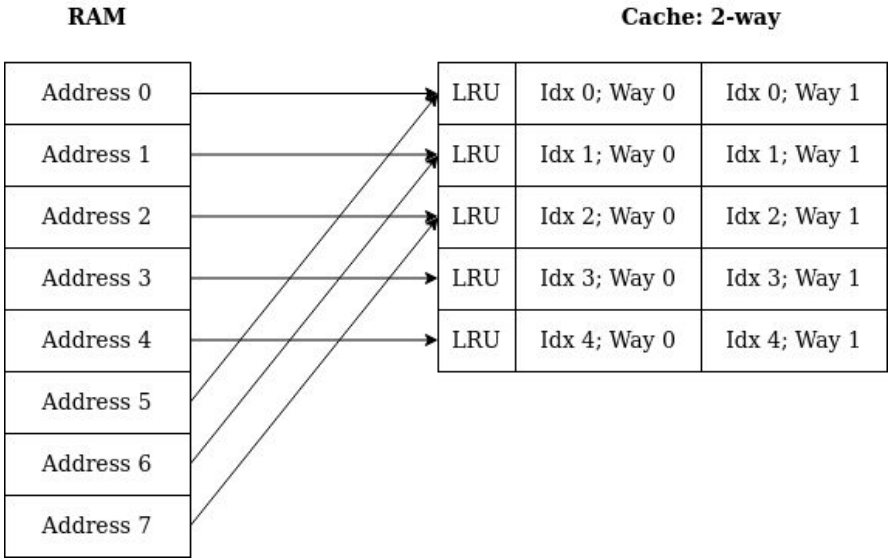


~~Reactive Programming~~ A quick not-so-off-topic #3

Cache structure: N-way set associative mapping
In the n-way set associative mapping the cache is divided into so called ways, that can be interpreted as yet another “collection” of values.

It uses an LRU bit, or set of bits, to specify which was the least recently used way.

Because n-way set associative cache allows for increased hit ratio it is frequently used. Keep in mind that because of its structure it requires more memory to keep the same number of entries.



~~Reactive Programming~~ A quick not-so-off-topic #3

Cache structure: Fully associative mapping

In case of fully associative cache mapping a memory address from the main memory can be mapped to **any** cache entry. This is convenient, indeed, but comes at a high cost. In order to fetch an entry it is necessary to check all entries in the cache.

Because it is time consuming modern CPUs usually either don't use fully associative mapping, or use it for very small caches, with a dozen or so of entries, or combine this kind of mapping with direct or n-way associative mapping.

~~Reactive Programming~~ A quick not-so-off-topic #3

Worst to best hit ratio, and the best to worst access time, can be seen below.

- Direct mapped (DM) cache – good best-case time, but unpredictable in worst case
- 2-way set associative (2A) cache, effectively 2x better hit ratio than DM
- 4-way set associative (4A) cache, effectively 2x better hit ration than 2A
- 8-way set associative (8A) cache, a common choice for later implementations, although in practice not 2x better than 4A
- Fully associative cache – the best hit ratio, but practical only for a small number of entries

See: https://coffeebeforearch.github.io/2020/01/12/cache-associativity.html

~~Reactive Programming~~ A quick not-so-off-topic #3

Somewhat less frequently used when optimizing programs is the concept of **branch prediction**.

Branch prediction means that a CPU tries to predict the path that the program will take in order to prepare future instructions. The problem is that sometimes it mis-predicts, resulting in (1) work done in vain to prepare next set of instructions, and (2) necessity to cleanup and load another set of instructions.

Recall from the table 4 slides before that a branch mispredict is about 5ns. In practical terms, to reduce the effect of branch misprediction make sure that the most frequently accessed code path is inside the if block, not the else.

The problem of branch prediction, and mispredictions, comes from the pipelined architecture of modern processors.

~~Reactive Programming~~ A quick not-so-off-topic #3

Some things that you must keep in mind when optimizing for memory.

1. Know thy caches: `getconf -a | grep CACHE`
2. Make the data small and try use arrays/contiguous memory chunks wherever possible
3. When doing low-level: align your struct attributes fit a cache block/page/line sizes (find it using 1.)
4. When doing low-level: align your heap-allocated data using `posix_memalign` or similar
5. Access data in regular patterns, preferably sequentially or strided
6. If necessary to re-access some data, do it ASAP
7. If having a way to pin threads/programs to CPU, do it
8. If possible, keep most frequent code in if block, not in else
9. Most importantly, always profile the code and find its bottlenecks

Reading list

- *"Concurrent Object-Oriented Languages and the Inheritance Anomaly"* by Dennis G. Kafura and R. Greg Lavender
- *"Actor Model of Computation for Scalable Robust Information Systems"* by Carl Hewitt (https://hal.archives-ouvertes.fr/hal-01163534v7/document)
- Elixir Getting started guide, Mix and OTP, chapters 2, 3, 4, 5 (https://elixir-lang.org/getting-started/mix-otp)
- *"Communicating Sequential Processes"* by Tony Hoare (http://www.usingcsp.com/)
- https://talks.golang.org/2013/advconc.slide
- https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/
- https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
- https://fsharpforfunandprofit.com/posts/recipe-part2/

Keywords (Good to know)

Software Transactional Memory (STM), Inheritance Anomaly,
Memory Thrashing, Lock-free algorithms
Systolic Array, Hypercube Algorithm/Topology, Superscalar architecture

Message Brokers

Alexandru Burlacu

Spring 2020

Message Broker?

Recall the PubSub pattern.

Now what if we have multiple publishers and any subscriber wants to subscribe to some of them?

Eventually it will become a mess. Recall from graph theory the bipartite graphs, and how many connections can be there if the graph is, God forbid, complete.

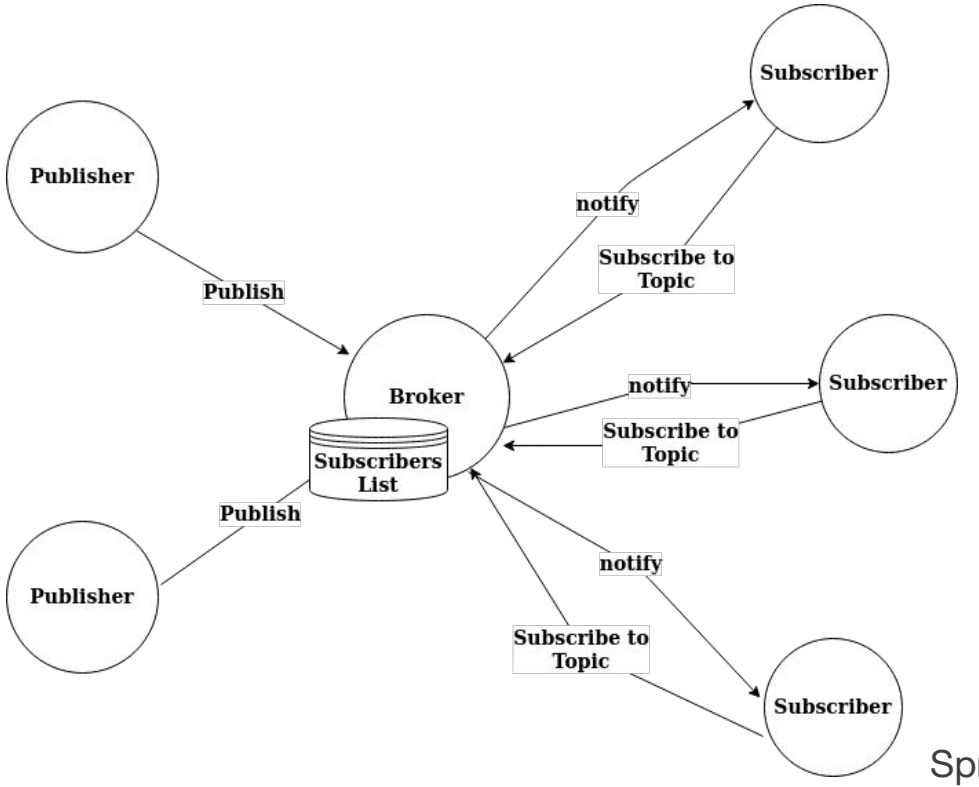
For 7 subscribers and 3 publishers we have at most 21 connections to manage, that's quite cumbersome. In practice, we may have hundreds on each side. Do the math and be scared.

Message Broker?

What if we can have an intermediary, like a post office?

Message Broker?

Enter the Message Broker!



Message Broker

A message broker is an architectural pattern that is most useful for (1) decoupling of sub-systems and (2) integration of different systems, each with its own messaging protocol.

If the first one is already clear, how about the second point?

Say, you have a couple of 3rd party APIs, some core business modules, one of which is legacy code, and the application logic. How can you make it all work together?

Message Broker

Say, you have a couple of 3rd party APIs, some core business modules, one of which is legacy code, and the application logic. How can you make it all work together?

There are 3 most common answers:

1. Have them share files. This is the most basic one.
2. Another way, maybe make it possible for the sub-systems to share a database.
3. And finally, let them pass messages.

Guess which one is considered best practice?

Message Broker

Now, you might be wondering, **why not write some adapters and have them communicate via HTTP for example?**

It turns out that in the end they would still pass messages, in a request-response fashion, so it's fine, sort of, because…

Secondly, point-to-point communication becomes complex pretty fast. Besides, recall, we also need adapters, and this means for each pair of message formats/APIs we will need separate adapters. Yuk!

Message Broker

So, message brokers. You can think of them as some sort of a chat for your applications. Just don’t take it too literally.

For a message broker to be useful it must enable its users at least the following functionality:

- Topics/Channels
- A way to subscribe and unsubscribe from it
- A clear message format (Preferably the message must be self contained, more on this later)

Message Broker

Note that message brokers are more of a concept, where's message queues are an actual implementation that is a message broker, just with a queue for the messages that arrive.

Another side note is the difference between a message broker and an event bus. Event buses are a way to use message brokers in a distributed system. Namely, an event bus is used specifically as a medium to transmit messages encoding events for services to react to.

For example an API service might emit to the Event bus a message of type UserLogedInEvent to which the newsfeed service will react by fetching the latest news and wrapping them into NewsFeedUpdateMessage.

Message Broker

Some modern implementations allow for much more than that, for example, Apache Kafka, which is many things, including a message queue, has distribution capabilities, durable message storage, and since recently exactly once message delivery semantics.

Other implementations might suport dead letter channels, for messages that couldn't be sent, message converters, very useful for integration use cases and other capabilities.

Message Broker

Recall the chat analogy?

Just as with a chat, a message broker usually is much more useful when it has a way to broadcast messages, or to be more technical, to multicast them.

The way to multicast messages for message brokers is to use so-called **topics** sometimes also called **channels**.

Message Broker

Topics are like group chats. And a topic always involves more than one consumer.

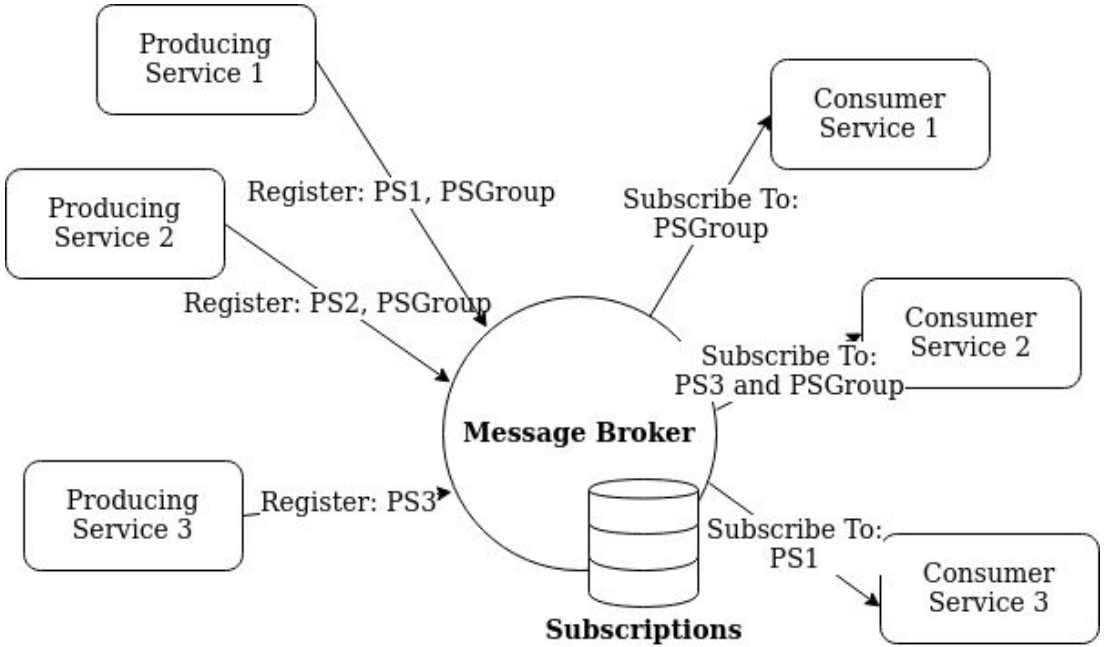
Think of it, you have a service that emits a message that is interesting for a number of other services. You can publish it into a topic and other services will subscribe to it, thus decoupling publishers from subscribers even more.

Message Broker?

Topics for Message Brokers

You can create new topics either on the producer or consumer side.

What if the topic is generated on producer side and there are no consumer services to read from it?



Message Broker

The answer to the previous question, as usually in SWE is: it depends.

If you don't care about those messages, just drop them. But what if you do?

Then you need to keep the arriving messages and send them over when an interested consumer subscribes.

This technique is referred to as **durable topics**, or durable queues, depending on the system and its architecture.

Note that durability does not necessarily imply written to non-volatile (like disk) storage.

Message Broker

Another use case: what if your broker crashes mid-processing the message, and you have a zero-loss requirement?

As a solution, store on disk the messages while being processed, a technique referred to as **persistent messages**. Keep in mind, this is safe but slow, because of the disk, duh.

You can put them into some file, database or maybe a log (like Kafka does).

Remember, **persistence** is about messages being written to disk, this helps if things go south while processing them. **Durability** is about the subscriptions, or rather, the messages that must be sent to the consumer but it is disconnected from the broker.

Message Broker

Now for misc. stuff.

1. Q: What if a non-persistent message is sent to a non-durable topic?
   A: If (1) broker fails or (2) subscriber is not connected, message is dropped

2. Q: What if a non-persistent message is sent to a durable topic?
   A: Message is lost only if the broker fails

3. Q: What if a persistent message is sent to a non-durable topic?
   A: If the subscriber is not connected, message is dropped

4. Q: What if a persistent message is sent to a durable topic?
   A: Congratulations, you have achieved zero-loss messaging… but did you need it?

Kafka, kafka, kafka… what's that again?

Apache Kafka is a message queue, but also an event/stream processing platform, but also can be used for storage, and analytics, and integration… it's a lot of things.

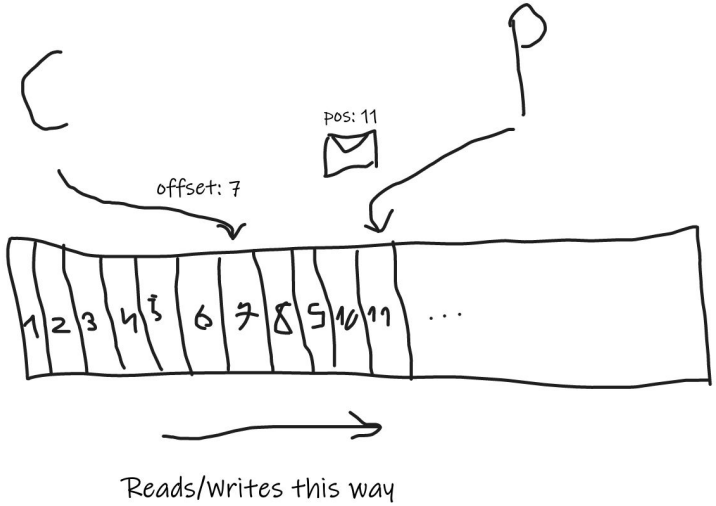
Apache Kafka is a bit different than "your usual message broker" in that it is not a simple relay, receiving messages and forwarding them to a specific consumer, but rather it is a persistent log which acts like a queue.

Kafka was built with scalability and fault tolerance in mind, and that drove the architecture towards the log-like structure and the possibility to partition data across many servers.

Kafka, kafka, kafka… what's that again?

Each topic in Apache Kafka has its own "log file" and can be **replicated** among multiple servers, to ensure that if a server crashes, the messages will be delivered anyway.

To keep track of which message each consumer has to read, they use offsets, like pointers to where is the next message to be read.



Kafka, kafka, kafka… what's that again?

In fact, each topic can be further splitted into multiple log files by some key, to keep the write ordered. This is called a **topic partition.**

A key can be some special property, or maybe some id/hash modulo the number of partitions.

Also, because of topic partitioning it is possible to greatly increase the throughput of the system. Regretfully, due to the persistent nature of Apache Kafka, it's not the best for low-latency communication.

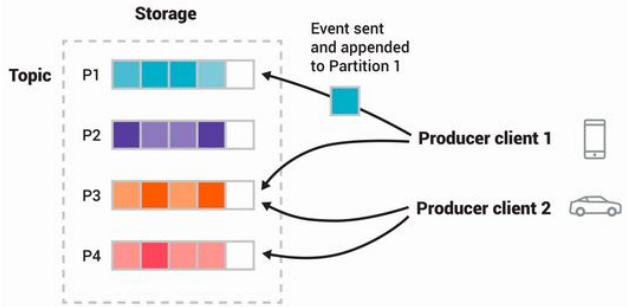


Figure: This example topic has four partitions P1–P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate.

Image source: https://kafka.apache.org/documentation/#intro_concepts_and_terms

Kafka, kafka, kafka… what's that again?

Events in a topic can be read as often as needed—unlike traditional messaging systems, events are not deleted after consumption.

Instead, you define for how long Kafka should retain your events through a per-topic configuration setting, after which old events will be discarded. It can be a couple of seconds or even a hundred years. This opens it for some exotic use cases, like using Kafka for analytics, or storage (See Event Sourcing).

Kafka's performance is effectively constant with respect to data size, so storing data for a long time is perfectly fine.

Message Broker - The Messages

According to Enterprise Integration Patterns book, which you are btw required to skim through for the exam, there are different types of messages, based on their intent, and some concerns about them.

- Messages could represent an action to be taken, a **Command**, the result of it, a **Document**, or maybe a change in the sender, an **Event**.
- Messages could be sent to be responded to, then you need to embed a **Return Address** in it, like with actors, do you recall? Also, keep in mind that the receiver is probably working with multiple messages at the same time, that's why you might also need a **Correlation Identifier**.
- Now, what if the response for a given message is huge? Take an inspiration from how TCP works and send a **Message Sequence**.
- Finally, what if the message is plain slow? What could happen if it is a Command Message and it comes to late? Having an **Expiration** date for it could minimize the impact.

Message Broker - The Messages

Everytime using messages as means for communication, consider making them **self-contained**.

But what does it mean?

Basically, make the messages contain all the information you need to make them actionable, so that you don't need to pass or keep some additional context or depend on some state.

The point of it is that you should strive to include sufficient information in the message so the state that is relevant to the current request is fully represented—removing and relocating relevant information should be considered a premature optimization until proven otherwise.

Message Broker - The Messages

For example, say you want to pass a document message *FetchedNewsFeed*, what do you think it should contain?

Should it contain the data representing the news feed? Or maybe it can use a reference to where this data is stored?

Also, should it contain the user for this given news feed?

Now, what if it's not a document message, but a command message, that is *FetchNewsFeed*? What has changed?

Message Broker - The Messages

Having an Expiration time for a message is considered good practice, especially when your messages are modifying some state.

Expiration is only useful when there are chances that some messages can be slower than others, usually something like this could happen if your message broker works in a non-deterministic (read concurrent) or simply some major network problem is possible.

Expiration won't, for example, save you from lack of ordered delivery of messages, or duplicate messages.

Message Broker - The Messages

Suppose you're sending multiple messages to some other service, and that it will reply to them as soon as it's done with a given message, thus, there's a chance the reply order could be different then the sent order.

Enter **correlation identifiers**. Simply put, it's an id value that is part of the message and which purpose is to re-identify the message.

Efficient usage of correlation identifiers allows for example to even reply to a message from a service different then the initial receiver of the message.

Correlation identifiers can be even used as means for tracing the messages inside a system, although this is an entirely different matter.

Message Broker - Few Patterns

Recall at the beginning of this chapter there was a statement about message brokers being an important integration tool.

The capability of messages brokers to work as converters, or **Message Translators**, is among the main features that enable them to the go-to integration tools.

Usually Message Translators can be either a plug-in component for a message broker or even a separate component in a bigger messaging system.

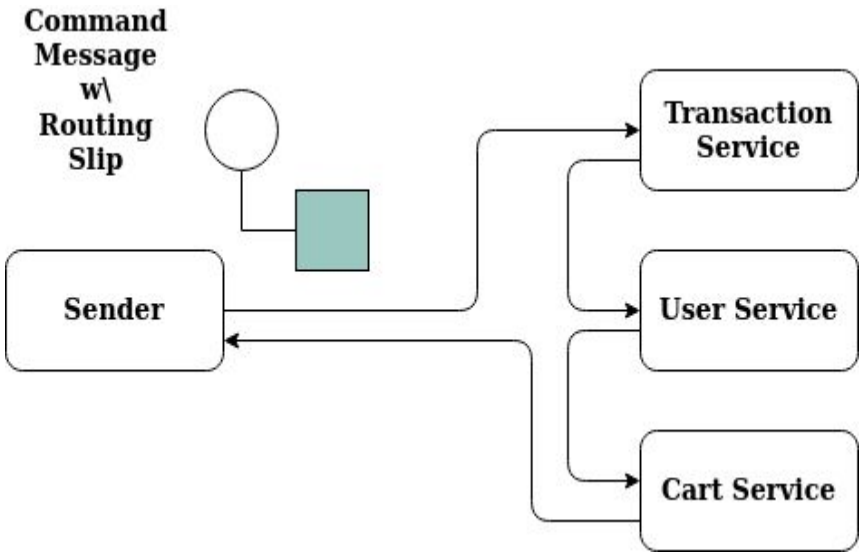
A message translator is the messaging equivalent of a GoF Adapter.

Message Broker - Few Patterns

Message brokers usually imply a bigger, message oriented system.

In such a scenario it is not uncommon to have a message sent to some service and it subsequently being transmitted further until at some point “replied to” by a service different than the first receiver.

In any case, if the transitions are known beforehand by the sender and encoded into the message, this type of behaviour is known as **Routing Slip**.

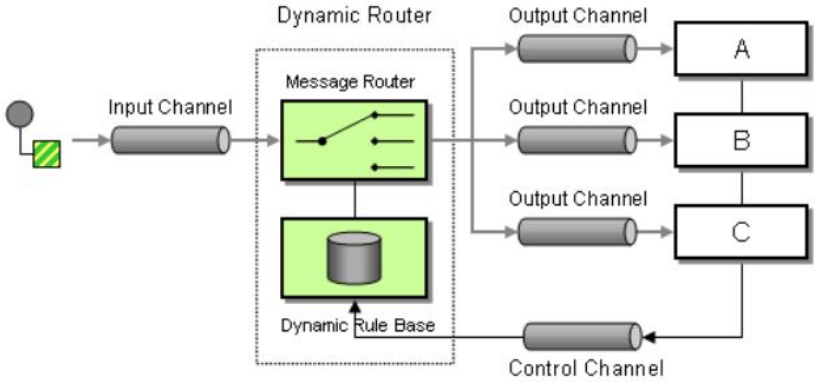


Message Broker - Few Patterns

Sometimes we may also want to change the routing rules depending on the current state of the system. We can use the **dynamic routing** pattern for that.

All subscribers in this case know about a special control channel through which they can send command messages to the router/broker such that the following messages are distributed differently.

One use case for this can be more fine grained load balancing.



Message Broker - Few Patterns

Another feature that is possible to use in combination with message brokers is to have **Content Based Routing**. Namely to have the broker decide where to forward the messages depending on the data that is there and not through a **topic/recipients list**.

Say the message broker receives the message that has XML data, that mentions payroll, then it will pass the message to the bookkeeping related services.

This feature must be used carefully, because it might result in a very big, fat message broker with numerous responsibilities.

On the other hand, it further minimizes the coupling between the services. Now they don’t even have to know the topics or services to which they are sending messages.

MQTT and XMPP

Alexandru Burlacu

Spring 2020

MQTT - Overview

MQTT, or message queue telemetry transport protocol is an open ISO standard, designed for inter-device, message-based communication that is lightweight, and primarily supports publish-subscribe pattern.

The protocol runs by default over TCP/IP, however, any network protocol that provides ordered, lossless, bi-directional connections can support MQTT.

It is designed for connections with remote locations where a small code footprint is required or the network bandwidth is limited. That’s why it is primarily used for IoT systems.

MQTT - Overview

The protocol was initially invented to monitor an oil pipeline in the dessert, and the communication link was via a satellite. It was 1999. After that the protocol was developed primarily by IBM.

For even more constrained environments there was developed MQTT for Sensor Networks, aka MQTT-SN that could swap the TCP/IP foundation with something else, like Bluetooth or UDP.

Such vendors as Azure and Amazon provide integration hubs with some support for MQTT. Just in case you seek to develop an IoT system.

MQTT - Protocol Details

First, you need to understand the high level parts of MQTT, namely the topics and the messages, after which some more advanced features will be explained to you.

One more thing, although it has MQ (message queue) in the name, MQTT does not provide message queue capabilities, only message broker ones.

Let's start with **topics**.

Generally, topics are topics, as in a general message broker. What you need to know is the naming convention. `home/kitchen/coffee_grinder` is a valid topic, the / are used to divide into subtopics. Also, topics are case sensitive.

MQTT - Protocol Details

Now, messages. MQTT, as of version 3.1, has 14 types of these. These are called control messages although they are used for data transfer too. An MQTT message can be as small as 2 bytes and as big as a couple hundred MBs. Server - **s**, client - **c**.

- CONNECT (**c2s**) and CONNACK (**s2c**), and DISCONNECT (**c2s**) to handle connection
- PUBLISH and PUBACK, and PUBREC, and PUBREL, and PUBCOMP for publishing, different kinds of acknowledgement depending on the QoS (later), also the message is bidirectional
- SUBSCRIBE (**c2s**) and SUBACK (**s2c**), to subscribe to some topic
- PINGREQ (**c2s**) and PINGRESP (**s2c**), to PING the broker
- UNSUBSCRIBE (**c2s**) and UNSUBACK (**s2c**), to unsubscribe from some topic

For more info check: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

MQTT - Protocol Details

A nice thing about MQTT are the so-called **retained messages**. These are normal MQTT messages that have the retained flag enabled, thus when they are received by a broker and should be forwarded to the subscribers of some topic, if there are no subscribers, the message will be persisted.

Only a single, latest, such message is allowed per topic, and it’s raison d'être is for the subscriber to receive the updates right away and not wait until the next message on the topic.

MQTT QoS

Each connection to the broker can specify a quality of service measure. These are classified in increasing order of overhead:

- **At most once (qos 0)** - the message is sent only once and the client and broker take no additional steps to acknowledge delivery (fire and forget).
- **At least once (qos 1)** - the message is re-tried by the sender multiple times until acknowledgement is received (acknowledged delivery).
- **Exactly once (qos 2)** - the sender and receiver engage in a two-level handshake to ensure only one copy of the message is received (assured delivery).

Most of the existing IoT hubs from cloud vendors do not support qos 2 and will either throw an error or downgrade you to the highest available qos, which usually is 1.

For a detailed overview, see: https://www.emqx.io/blog/introduction-to-mqtt-qos

XMPP - Overview

eXtensible Messaging and Presence Protocol (XMPP) was primarily developed for chat systems, but eventually became one of the most used protocols for IoT systems. It has also been used for VoIP and gaming.

Just like MQTT it was initially developed sometime around 1999, and was known as Jabber protocol.

XMPP uses XML for messages. Basically it streams chunks of XML.

XMPP - Overview

XMPP is primarily used for instant messaging (IM) and because of that it provides not just messaging but also presence capabilities.

Presence means the capability to see the status of your contacts. Think Online/Offline/Away/et cetera in Skype.

Because of XMPP's native support for XML it is an extremely good protocol for machine-to-machine communication and software integration.

XMPP - Protocol Description

The basic unit of communication for XMPP is the so-called **stanza**, and there are 3 of these: <message> - this one should be self-explanatory, <presence> - used to inform the status of the contact and <iq> (Info/Query) used to either get information or apply some settings.

```
<message
     from="SenderName"
     id="SomeId"
     to="ReceiverName"
     type="groupchat/chat/normal/error/headline">
     <body></body>
</message>
```

XMPP - Protocol Description

There are a number of types of message stanza:

- **normal**, which is a message that is sent without it being part of a chat
- **chat**, a message meant to create or continue a p2p chat
- **groupchat**, think of it as multicast, or, as a group chat
- **error**, in case of some error
- **headline**, is a broadcast message, like an announcement, or news

Headline and error are not expected to be repliable, while normal, chat and groupchat are.

XMPP - Protocol Description

A message stanza also can contain a number of children tags, namely a <body>, which is mandatory, but also a <subject> and a <thread> children are possible.

Threads allow mail thread-like capabilities, or Slack-like reply to a message in the channel.

The <subject> tag can be used to model topics, and it is allowed for a message to have multiple subjects.

https://xmpp.org/rfcs/rfc3921.html

XMPP - Supported Patterns

XMPP supports Publish-Subscribe, Request-Reply and Asynchronous Messaging.

XMPP is like email, anyone can become a server and communicate with everyone else.

XMPP is eXtensible, remember? Therefore there are XEP or XMPP Extension Protocols, that allow support for monitoring, working with sensors and other perks.

XMPP doesn’t have Quality of Service and it is text-based. Therefore not the best choice when resource constrained.

Reading list

- http://zguide.zeromq.org/page:all
- “Enterprise Integration Patterns” - Gregor Hohpe and Bobby Woolf
  https://www.enterpriseintegrationpatterns.com/docs/EDA.pdf

Keywords (Good to know)

AMQP Protocol, RabbitMQ Architecture
ZigBee, Constrained Application Protocol (CoAP)