# Distributed Systems

Projects are a poor model for software development, long-lived products are better

## Systems are not Applications

connectivity → the network matters

- reliability — data can be lost when sent over the wire — acknowledges, retries, caches, ...
- latency — it takes time to cross the network — wait, expire, ...
- bandwidth — network congestion may interfere — discarded packets, time-critical data, ...
- security — bad actors on the Net — threat model analisys, risk analisys
- topologies — resources can go down or move to another subnet — DNS, dynamic routing, ...
- administration — updates, dependencies, ... — logging, monitoring, ... auditing, journaling, ...
- transport costs — serialization / deserialization
- diversity — lots of different systems, languages, architectures, ... working toghether

"If you have to cross the network, take all the data you might need with you."

- (several steps), non-atomic operations
- (continuous) evolution
- (decentralized), distributed logic

## Coupling

- platform — A works with B
- time — processing time of B affects that of A
- spatial — where is B ?

## Decoupling

use text based standard representations and protocols — JSON, XML, ... — HTTP, SMTP, UDP, ...

use messages, publish/subscribe, ...

delegate communications to lower layers, load balancers, proxies, ... queues, ...

## Messaging

Services coupled to messages, not to each other

- asynchronous messaging → one-way, fire & forget messages
- strongly-typed messages
- represent methods as messages
- include id, version, timestamp in each message → ordering issues
- include relating info, in order to correlate... what had happened ?

## Processes

process : a set of activities that are performed in a certain sequence, as a result of internal or external triggers

long running processes : are stateful

use Sagas to manage the flow
use Adapters to manage integrations

## ① The network is reliable ?

it works most of the time ...
... except when it don't work !

- our code must be aware of timeouts, broken links, congestion, ... on the sending way and on the returning way
- in asynchronous scenarios we can try to retry the operation or to recover by other means in the background
- but in synchronous scenarios we must take into account that the user is waiting for the response (response time matters in Request/Response scenarios)
- in distributed systems, with many actors involved, with several steps, there are too many edge cases when we try to recover from a failure in one of the steps

transactions integrity is tricky in distributed systems

can we build a reliable system on top of a fundamentally unreliable network ?

the response to this question is "yes, but we are to going to build it differently than the traditional Request/Response pattern we are used to use on desktop applications

better to use a MQ (Message Queue),

use a reliable messaging infrastructure and adapt(*) the architecture of the application / system to work with this paradigm

(*) we lose Request/Response kind of operation

## (2) Latency is not a problem (==0) ?

time to cross the network in one direction
- is small for a LAN
- can be large for a WAN

it is many orders of magnitude slower than in-memory access

1 (sec to year) $10^0$ — $10^6$ (ms to ms) $10^{-9}$ — $10^{-3}$

Don't do use remote objects!
avoid them as much as possible

don't chat back and forth with a remote object

Don't cross the network if you don't have to.

Use DTOs, and take all data you need once, and cache it locally for further use, whenever it is feasible to do so

send & forget style of data interchanges

Inter-object chit-chat shouldn't cross the network, do it locally.

"If you have to cross the network, take all the data you might need with you."

but beware with how this affects to the bandwidth limitations on your networks

we need to have more than one domain model to resolve forces of bandwidth (lazy load) and latency (eagerly fetch)

DTO (Data Transfer Object)

See next page back (3) ↩

## (3) Bandwith is not a problem (∞) ?

how much data you can move through the network, is going to have severe ramifications on the performace of your application

don't understimate the bandwith of a truck full of hard drives going by highway

(moving bytes accross a network is much much slower)

Although bandwith keeps growing, the amount of data to trasfer grows faster.

| 2000 | 2024 |
|---|---|
| 1 gbit ethernet | 10 gbit ethernet |
| 1 core | 8 cores |
| 2 gb RAM | 64 gb RAM |
| 512 gb. disk | 2 Tb. disk |

1 gbit ethernet ≃ 125 Mbytes/sec

Tcp/ip headers and coordination take half of it

≃ 64 Mbytes/sec

data serialization format can take another half

≃ 32 Mbytes/sec

If you have network congestion problems, putting more servers will only worsen them

congestion problems lead to latency problems, and latency problems lead to timeout problems

problems ⇒ retries ⇒ more problems

you need a more efficient way to access to the data

(or a network with more bandwith ;-)

or a staggered access to the data

or several, different, independent, routes to the data

## ④ The network is secure ?

The only machine/system that is secure
is one that is disconnected from anything else ...
turned off and buried in 3m of concrete

we, human beings are the weakest link in any security chain. social engineering is the best tool of a hacker.

nowadays, with practically all things running in the cloud (Internet), and or connected to the the systems becoming more and more complex and interconnected, the ability to build systems that are secure is more difficult

You can't be 100% safe from everything

So, perform a thread model analysis and balance costs against risks

- what is the cost of protecting something?
- what is the risk (damage) if this something is compromised?
  or what is the advantage (benefit) someone will gain if compromises this something?

But, in first place ... is this way of connection necessary?
(attack vector) (convenient)

The best hacker proof door is the door that doesn't exist.

## (5) The network topology don't change ?

Programming models that can run into issues in environments where the machines and networks are more volatile than you assume.

for example - WCF callbacks contracts

- config file values for machine ids
- clients or servers that disconnect and dissapear from the net after subscribing or registering themselves with a service or pool
- etc

Be extra aware of fast-dirty-demo code that nevertheless ends-up deployed in production.

- Don't hard-code addresses
- Consider using resilient protocols (multicast)
- (Auto) Discovering can be cool, but it is hard to get it right.

Test your system regularly to know how it performs under changing topologies (for example one or several instances of a server goes down, or reboots at different times, or try to reboot before some other service is available, or ...)

chaos monkey tests

## ⑥ The admin will know what to do ?

The all-knowing admin that maintains flawlessly the system is only possible in small networks. And if so, only until he/she got promoted and leaves...

Will the system survive to changes in working procedures, upgrading policies, new versions, new rules, ...

Make clear any dependency between different parts of the system. Or better, make each part as independent possible

The more configuration options we create, the less likely that anyone knows what a given set of specific settings is going to do

Do invest in some kind of automated configuration management

Take time to slow-down and document

Take snapshots of well-know working points and be able to return to them if something goes wong

Do invest in some kind of loggin/monitoring management

Know how looks your system when it is performing well, to detect problems when they start to show-up

The more moving parts you got, the more likely that some go off of synch with each other and something not going to work

The more and more code we write, the more changes we make, bigger the chances of faillure at the next upgrade/deployment to production

CI/CD

and automate as much as possible of the pipeline

## (7) Transport cost is not a problem ?

Serialization before crossing the network takes time(*). And the same deserialization on the other side.

(*) it takes a bunch of CPU time

And usually goes unnoticed, behind the frameworks we are using in our code.

Physical architecture

It is cheaper/faster as more monolitic is our system. (for example, reading all from the same ~~machine~~ memory is cheaper/faster than retrieve all over the network from different machines)

Logical architecture

It is more flexible/scalable as more modulars and more independet those modules are.

Each architecture is better suited for a certain kind of application.
Is our work to find the most optimal efficient architecture style given our requirements and constraints.

All is a trade-off balancing the cost perspective, the complexity perspective, the maintainability perspective, the scalability perspective, ...

connections costs tend to grow exponentially as we go to larger and larger links. Some architecture can be feasible inside one unique datacenter, but can be infeasible if distributed all around the world.

## ⑧ The network is (interoperable) homogeneous ?

Some years ago (2005), there was only the interoperability between major languages (aka .NET ⟷ Java)

But nowadays (2020) the decentralization trend forces us to manage between different languages, different database styles, different API styles, different encoding formats on different systems hooked to work together

- languages: Go, Ruby, Javascript, ⋮
- database styles: Relational, NoSql, Graph, ⋮
- API styles: REST, RPC, Webhooks, ⋮
- encoding formats: JSON, XML, YAML, ⋮

Semantic interoperability will always be hard, budget for it.

for example: We don't know what to put in place of missing mandatory fields for a system that requires them when the data reaches us from another system that doesn't require them.

Be aware that almost nothig works out-of-the-box in the interoperability world

You always need to make adjustements, from small tweeks to big integration rewrites

- Take into account that things can fail on a network or change

- Watch for latency and bandwidth issues

- watch for security issues

- Automate as much as possible and deploy
  - configuration
  - integration (CI/CD)
  - deployment (CI/CD)
  - monitorization

  } DevOp operations

- Avoid as much as possible interoperability scenarios.

- Avoid as much as possible network communications specially the long-distance ones.

- Allow as much as possible logical modularity and minimize as much as possible the dependencies between modules

The system is atomic ?

BBoM (Big Ball of Mud) → a program that had evolved into a logical monolitical monster

can be worse → if it is physical distributed !
(diferent parts of the logical monolit running in diferet machines

logical monolit : all parts of the code tightly coupled with each other

logical distributed : code organized in well defined modules with clear interfaces between each other

physical monolit : all parts running in the same machine

physical distributed : different parts running in different machines (linked together trough network connections)

scalability { Vertical : bigger, more powerfull, worker
horizontal : more workers

Beware of false "services". A service is an indepedent part of our system that performes a substantial work on its own, without need of other parts to do so.

If a "service" needs collaboration from a bunch of other "services" to get its work done, all them are one logical service (one logical monolit)

A logical distributed program can be deployed on a physical monolit manner or on a physical distributed manner.

But a logical monolit program can only be deployed in a physical monolit manner (can only scale Vertically)

(try to scale horizontaly - distributed - leads to worsten performace - network penalty -)

## The system is atomic (cont)

How do a program turn into a BBoM ?

Some forces that can lead to this outcome :

- integration through a unique database
  (all code is tied to that database)
  (coupled)
- patch other supplementary data sources
  on top of that unique database
  (usually because we cannot chage the schema
  for fear to broke something else)

If you are saying "we don't be doing this...",
"we must fix this later,.. when we got time..."
You are onto the perilous path toward a BBoM

Please, stop! and take time to refactor
the system to maintain it's architectural
stability

---

A monolitic data architecture cannot scale horizontaly !

It is not a easy task !

Horizontal scalability is only possible if
our system is designed to allow it.

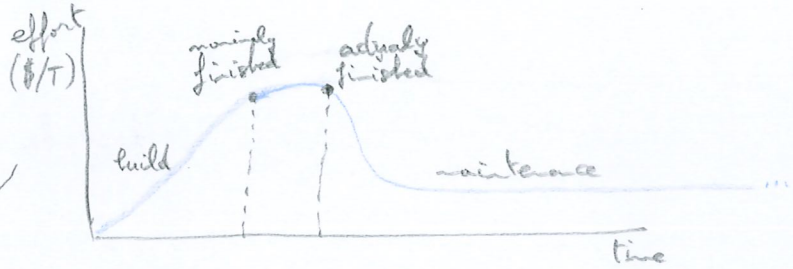
Taking into account the logical boundaries (if they existed, of course)
in our model, and leveraging the system
to take advantage of them.

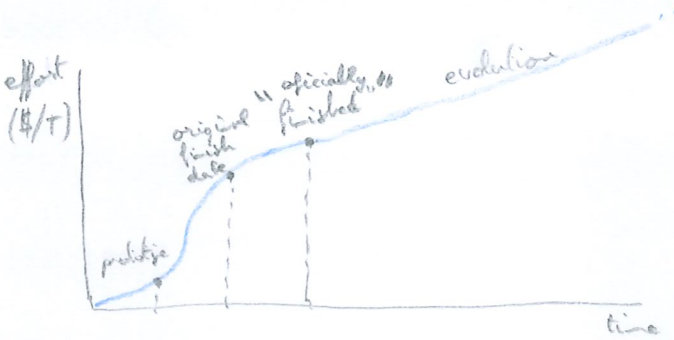
Independent services can run on independent machines. We can raise several nodes with a service
as we need more power for this service

# The system is finished ?

There is not "maintenace" phase in a software project.

A software system is never "finished",
it is more like a long-lived product
in continous construction/evolution.

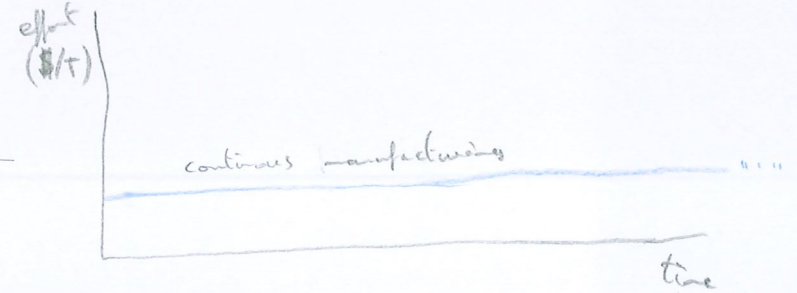



In other kind of projects and products,
the kind of activities performed and skills
needed to perform are quite different on
design/build phase than in maintain phase.

But in software systems they are the same,
always design/build ones... all lifetime of
the system.

To talk to the business people: software is a product not a project




launch a product takes care of the long-term (viability) manufacturability of this product

## A better development process ?

When asking for new software / new features
Business People don't give requirements.
they give workarounds that they through will allow the system to do something they need to do. what

IT Business Analist must have sufficient
"gravitas" or "political capital" to be able
to say "lets have a conversation" to discover
and analize what is actualy needed,
why it is needed and how to achieve it.

IT Architect must have sufficient
experience to be able to give
some estimates, try some POCs (Proofs of Concept),
and refine the estimates, until all
stakeholders agreed to some requirements
and associated costs. (requirements are an investiment decission)

the Holy Grial ?
colaboration / communication between
all people involved,
Business and IT,
until they understand each other
and agree on something
realistic enough to do

"Given a well-formed team of N persons,
that is not working on anything else,
I'm C% confident work will take between T1 and T2 months / milliards € / ... "

A well-formed team is a group of people that had worked together in something similar.
It is not a bunch of people just assembled for the tasks

## Business Logic can and should be Centralized ?

When we implement something in three or five places all around the system .. What happens when business rules change ? we risk to forget to change one of these places !

But, if it is implemented only once, in a given component . then we have other two or four components that depend on this centralized component

What happens when business rules change, but they change differently for each of those components ?

we need to walk all those components to broke it's dependency and rewrite for the new rules

Architecture requires more than one point of view to grasp all it's complexity , all of it's different aspects of a ~~living~~ system

You can tag source code by feature implemented, (the pull request or commit that implements/changes the feature)
In a manner that enables you to find later by feature , even if it's in multiple files/places.
So you don't need to implement it on a single place, avoiding dependencies.

has references (tags) to all places this feature touches

Duplication isn't the devil !

in more loosely coupled codebases is less likely that a change will going to break something elsewhere

DRY → has evolved to don't be DRY, we WET :-)

Don't Repeat Yourself

Write Everything Twice

measuring degree coupling is a very subjective matter

- what kind of coupling ?
- how much of each kind ?

## Coupling between two parts of a system:

- **Afferent** (incoming) : who depends on me ? → too many signals that a change in this component can potentially break a lot of things elsewhere
- **Efferent** (outcoming) : what are the things I depend on ? → too many puts in doubt that this component fulfills only a single responsability
- **Platform** : too specific or proprietary libraries, protocols, ... that are only available on one platform
- **Temporal** : processing time of one component affects processing time of other component
- **Spatial** : only one specific machine can handle the calls process execution once it is initiated on that machine

Take into account when business requirements demands consistency

Beware of shared resources (DB, API, ...) that hide coupling between components that share them

---

To minimize platform coupling (and improve interoperability) , we can use
- **text based** representations of data (xml, json, ...)
- **standards based** transfer protocols (HTTP, SMTP, UDP, ...)
- SOAP services, REST APIs , ....

don't reinvent the wheel !

To minimize temporal coupling ,
- first be sure that the process is not temporaly coupled (for exemple if component A cant proceed until component B provides usefull data) (cannot do any sensible work)
- avoid repetitive query ask for some kind of data , **cache** it when you receive it in the first query
  - if you can use stale data

To minimize spatial coupling

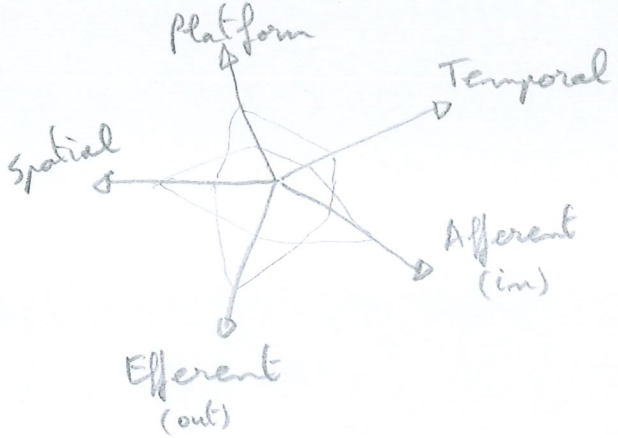
if you publish data, state its validity

- **publiser / subscribers**
  - only one logical publisher for a given kind of event

don't publish commands
publish only facts

Don't apply technical rules based on metrics . Persons have a tendency to hide things to make the metrics look good , instead of actualy solve the problems that the rules try to watch

Coupling is a function of 5 different dimensions:



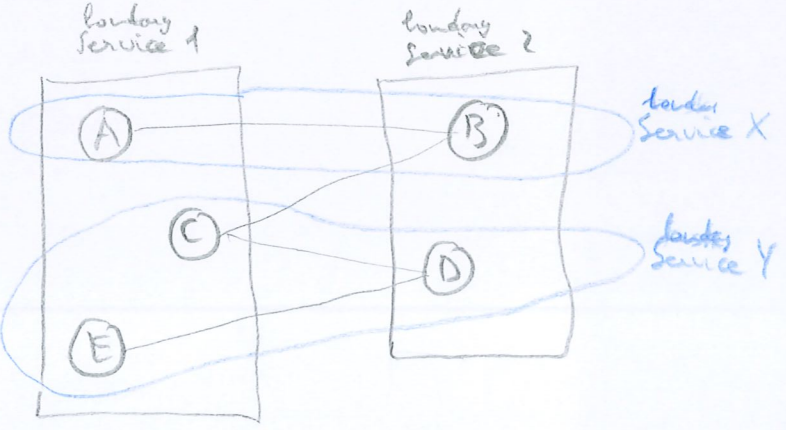

Each technological or architectural decision we take, position us on different parts of this diagram.

Coupling is not loose or tight,
the question we must ask ourselves is
"is this coupling situation reasonable in this context ?"

There is no "silver-bullet" for all,
There is no "free-lunch".
All has trade-offs and equilibries

Clear boundaries with clear responsible relationships between them.

Beware of tasks that cross boundaries at several points. They are a warning signal that some boundaries need rethinking.



Asynchronous messaging ⇒ { one-way, fire & forget } messages

Ⓐ launch something

Ⓑ continue doing sensible work

Ⓒ when response arrives (latter) or when it is convenient for us, continue with the work that needs something from Ⓐ

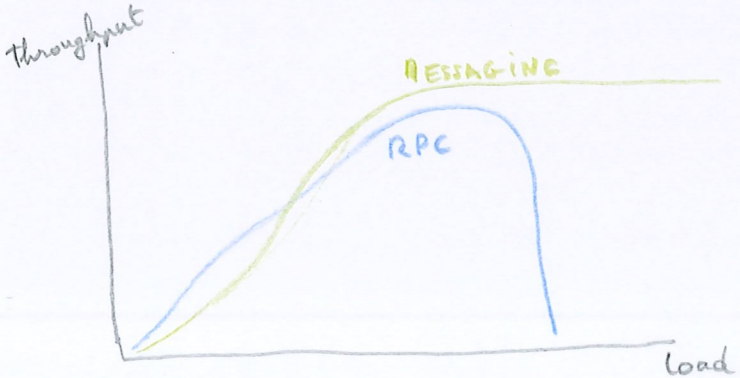
If our task at hand doesn't have Ⓑ (we must wait hand-on-hand until, doing within useful until Ⓒ)

The task is synchronous and not asynchronous! and blocking

Each message has an id

Store & Forward adds reliability

RPC vs MESSAGING




RPC (Remote Procedure Call)

has blocking calls and have no time to free memory

With MESSAGING, threads are independent

The remote server start rejecting requests → and the system go down terribly

to process much much faster...

The Queue can continue storing messages to disk

## Service Interfaces

- public functions in a library or similar
- public remote calls to an API
- queues or topics where to send messages

→ - command-centric
or
→ - document-centric (resource)

## Messaging interfaces

- messages of certain types, with a certain bunch of attributes / properties / fields (they are as DTOs)
- handlers for each type of message

In order to distribute / share message definitions, package managers become an important tool to manage artifacts in a distributed system. (packaging and versioning)

The cross-cutting stuff (security, logging, ...) usually goes transmitted on the headers of the queuing transport system and not in the body (the payload) of the messages

On top of the messaging delivery system, you articulate the business logic architecture:

- Event-driven
or
- CQRS (Command-Query Responsability Segregation)
or
- Microservices
or
- etc

also
DDD (Domain-Driven Design)

# Fault tolerance

- when servers crash
- when deadlocks happen
- when network are saturated / down
- when databases are saturated / down
- ...

If failed, you need to log / store the data or you will lose that data

direct calls to APIs need retry, acknowledge, timeouts, ...

message systems simply stockpile the messages to be processed latter, when the database / server / network ... is available again

store & forward mechanisms receive and store messages in a place (same machine) to be sent to another place when possible (across the network)

Log

Write a message in a queue is a small / fast / reliable operation.

The messages stay in the queue until the system can process it.

It is easy to transfer messages from one machine to another.

No business logic, database transactions, ... are involved!

There is a record of the data / action waiting to be performed!

or from one queue (for exemple: pending) to another (for exemple: done)

And also saves as a log that can be monitored or audited

After some number of retries, if message processing fails, usually, it is tagged as 'poison message' and put in an error queue for "failed messages" (adding the appropriate information about the failure reason) (with the appropriate notification to someone in charge)

When the problem is solved, you can retry the failed messages. → you don't lose data!

→ you have rich monitoring data to help troubleshoting!

# Auding / Journaling

A message queueing system is also a "running log" of what had happened and what is happening in the system

Messages can be saved to a long-term storage after they are processed.

So, you can have a central log of everything that happened.

But it can be difficult to interpret by itself.

"Related to" fields are very usseful when tracing what had happened. Helping to correlate one message to another

Usually they are based on "Message id" fields

in order to trace the flow of messages related to a given action

"Timestamp" fields are very usseful also. Helping to correlate temporally the flow of different queues. different sources.

## Calling external web services

If one step fails ?

and you need to rollback ! the steps already done ?

Weird things start happening (eventual inconsistency) when you mix different paradigms (messaging, RCP, REST, ...) in the same endpoint processing code.

DTC !

Distributed Transaction Controller

Distributed transactions are very very tricky, very very difficult to do well. Avoid them whenever it is feasible to avoid.

ACID consistency is very difficult to attain when working with distributed systems !

It is easier to do with "user generated id"s (for example UUIDs). → you have all steps identified with the same id

It is difficult to do with "database generated id"s → first steps have a different id, or nothing can be done until the database provides an id for the first step of the process

or you must hold lock certain steps until you have the final database step and you can retake those steps and complete them ... but ... what if any of them fails ?

It is impossible to do if the ids must be correlative and there can not be gaps between ids. (When a process fails, others processes had advanced the id counter, how can you redefine the failed id, ? .... !?! ...)

monolitic systems
layered | services systems
messaging | events systems

} different kinds of architectures (hamers, screwdrivers, ...) for different kinds of needs (nails, screws, ...)

---

organizational inertia is not on your side,
organizations want to keep doing what they always being doing.
Any type of change you are proposing, you have to push it up-hill.

---

How to reduce do-now real-time chunk of process (synchronous part)
and expand do-latter chunk of process (asynchronous part)
so you can cope with big load peaks

- Save to database (synchronous part) and batch process (asynchronous part)
- Save messages to a queue (synchronous part) and read from the queue (asynchronous part)

The async-effect is that is cost effective,
not the messaging-effect

(messaging systems are intrinsically async)

---

With messaging systems, we can separate the do-latter part on several machines.
With batch processes, we need to use the same database; and it can became a bottleneck.

But, the several machines ultimately also need to use the same database.
if you don't include all the data you need to process it in each message.

- don't do value judgements like "X sucs", because they are easily counter with "Y sucs", ... "Z sucs", ... all all sucs at the end

- don't appeal to figures of authority, because anyone of us has his/hers loved autority figures (instead, we can try to reach to agree on who can be a figure of authority / reference on wich all of us agree)

- ground on tecnical arguments explaing benefits for the situatio we are talking about in each moment.

- instead of "— is more —" try to explain "why — is more — in this situation", "how is — more — in this situation"

Messaging systems kind of forces us to think upfront on separation of concerns, temporal and spatial coupling, ... and other aspects of distributed systems.

But this guarantees nothing. We can do a "messaging mess" as we can do a "call services mess" or a "call functions mess"

A poor designed architecture is a poor designed architecture, regardless of the technology it is built in.

---

Start small, break things, do mistakes, ... and learn.

---

Start with applications - that are highly async: send email, generate PDFs, ...
- that have basic batch processing processes

---

Little by little - developers are gaining experience with messaging technology
- business are gaining confidence with the new technology
- and system admins

---

learning, training and willingness to understand new ways is key!

- potential benefits only realize themselves if we have enough developers that had gone through enough of the mistakes with the new technology to be able to figure out what is the right way to apply it.

- the issue do not reside into the technology, it is into the architectural mentality the developers had. You can do right monoliths right distributed with practically all any technology.

(And so for bad monoliths, bad distributed)

# Distributed Communications Patterns — Long Running Processes

How to manage the state of long-running distributed communication?

Orchestration

Sagas

It is not about time, is about complexity of state:
- regular processes → basic process control: if/then
- long-running processes →: state machine

In long running processes multiple external events/triggers/calls are handled by the same process instance → so, it is stateful

multiple steps that change state, until we arrive at the final state → endProcess

each step non deterministic, the process must wait until the step is completed; and then, proceeded with the next step that leads to the next related state

---

Message Handlers → process a message, and done! (or failed...)

Saga Handlers → a message triggers the process, the saga ~~messages~~ has an internal state, and works until the process arrives to its final state (or until it fails...)

internal state at each step must also be persisted

a consistent state that can be persisted

and we must rollback the side effects any of the performed work steps had caused

avoid as much as possible distributed transactions !!! (distributed long running processes)

this is the difficult part of any transactional process

and more difficult in distributed transactions

In a saga, avoid to alter directly (to query directly) the master data during processing each step.

A saga is about to process all the steps,
persisting its internal state at each one,
until it completes the process and
saves the final result. → usually not directly, but by invoking some other services or sending messages to tell others to do something

If the processing fails, nothing is altered.

And we can retry the saga.

Rollbacks can be performed:
- reverting the changes
or
- making new compensatory changes

The principal responsibility of a saga is the orchestration of the work to be done.

---

In multipart business cases
~~If the business case is complicate enough~~, sometimes
we cannot solve/orchestrate them by technology only.
There are also involved business decissions/contracts about
how each part must behave on different situations
and what each part can (or not) expect to do/receive.
(level of service agreements)

doing queries as a part of commands
over history in a highly consistent fashion

→ the "Holy Grail" for distributed transactions

In order to have very encapsulated (autonomous)
objects that do not depend on others.

Do as few request/response outside it
as possible, preferably none.

But for that, this object has to have
inside it all data it needs to work.

The easy part is using the building blocks.

The hard part is analyzing the
business processes to identify
what the steps should be.
And what the boundaries are for each component

In multistep processes
When interacting with external services / 3rd party systems / legacy systems,

- use a saga to manage the flow
- use a separate adapter for each integration