ERLANG FACTORY SAN FRANCISCO 2016

Erlang
SOLUTIONS
EST 1999
SAN FRANCISCO LONDON STOCKHOLM BERLIN KRAKOW BUDAPEST BUENOS AIRES SEATTLE

HELLO!

Hernán Rivas Acosta

hernan@inakanetworks.com
www.erlang-solutions.com www.inaka.net
@inaka

inaka

Erlang SOLUTIONS

www.erlang-solutions.com

Making a reliable reporting system with Kafka

Choosing the right **tools**

tigertext REQUIREMENTS

- Lots of instances and microservices producing reports at the same time
- Messages can not be lost, this logs are used to provide statistics and audits
- Unknown number of simultaneous consumers
- Having the messages in order is prefered

The options

Flume RabbitMQ Sparrow Starling Azure

ZMQ Kestrel Kafka ActiveMQ

SQS EagleMQ Celery

WHY KAFKA?

- High throughput and low latency
- Used by:

- Redundancies built into the system
- LinkedIn talent

WHAT MAKES KAFKA SPECIAL?

- Distributed architecture
- The concept of **topics** and **partitions**
- The replication factor, offering redundancy
- The performance

Implementation

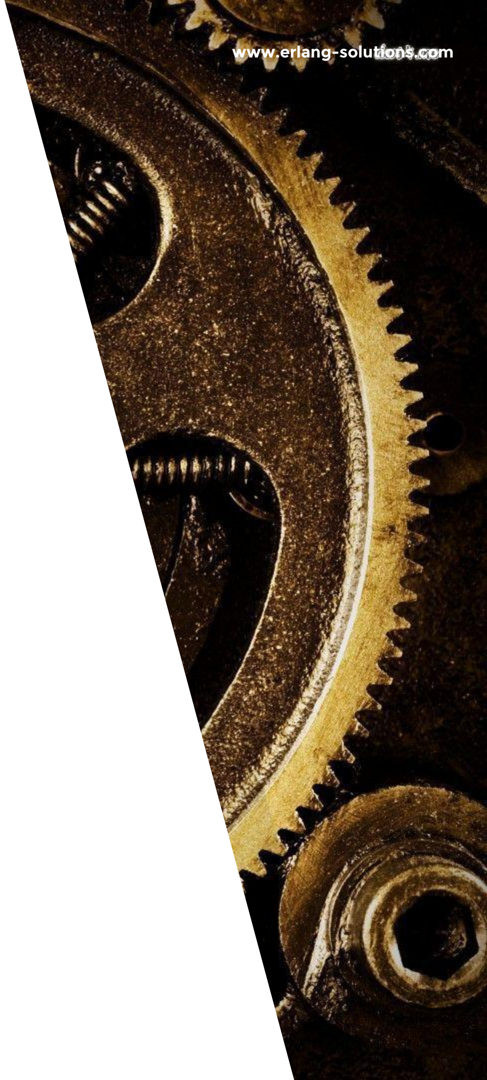
The Erlang side

OUR OBJECTIVE

Support all features that made us choose **Kafka** initially

LIBRARY REQUIREMENTS

- Reliable
- High message volume
- Minimal performance impact
- Lose no messages
- Maintain message order whenever possible



www.erlang-solutions.com

The libraries

wooga/kafka-erlang

Great company!

The protocol is tested

Buffers the entire response in RAM before parsing it

No support for 0.8 (the latest version)

klarna/brod

It’s a really good name

The protocol is tested

Well documented

Serializes all messages through gen_server so it will not handle message bursts as gracefully

The consumer buffers the entire binary before attempting parsing

helpshift/ekaf

Excellent documentation

Currently being maintained

Code has lots of comments and it’s pleasing to the eye

Using a FSM is a good choice

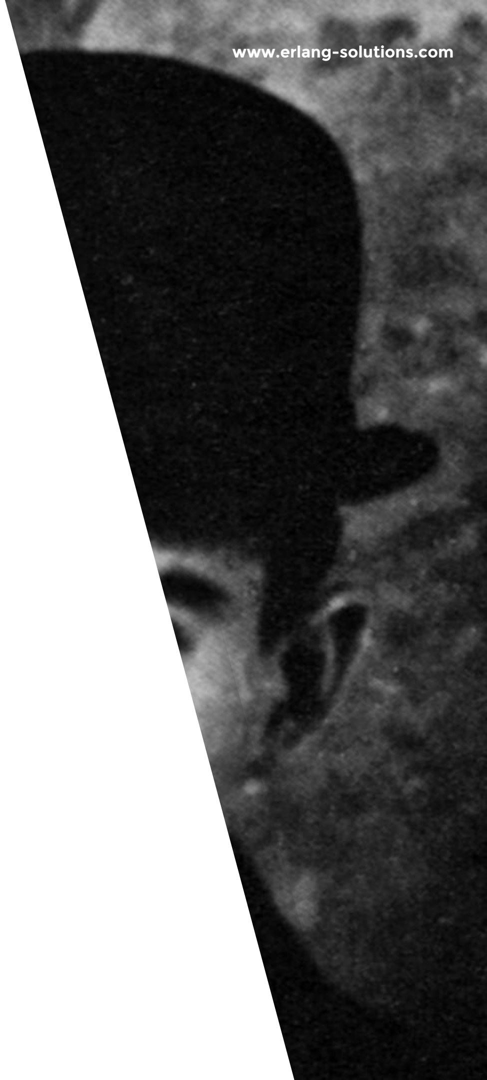
Suffers from some of the same issues the previous libraries have

It has no consumer

Kafkerl

HernanRivasAcosta/kafkerl

- Fast binary creation
- Request caching for better use of bandwidth
- Highly concurrent
- Messages are not lost
- Handles all server side responses
- Can parse and consume partial messages
- Simple API
- By design, it supports for all Kafka features we needed
- No connection to Zookeeper



How?

AVOID SERIALIZATION

Avoids serialization by storing the messages on **ETS tables**.

Provides as much **concurrency** as the system it's running on has.

More on this later.

MESSAGE PARSING

Parses **partial responses**. The entire response binary is never stored.

How?

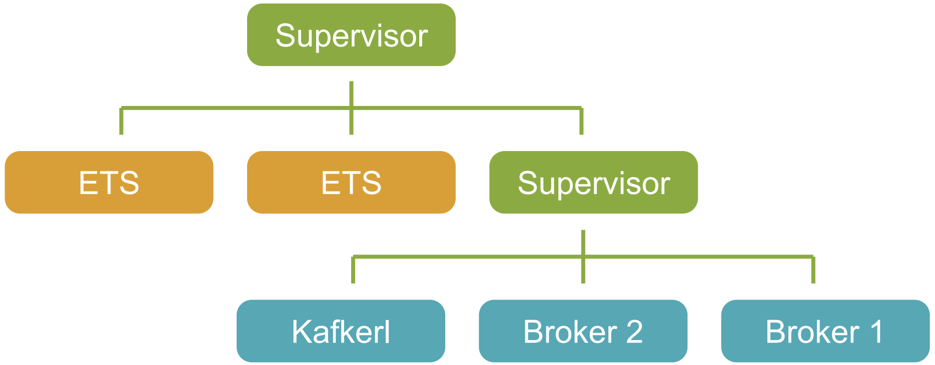
```
-type state()    :: {binary(), integer(), [any()]} | void.

-type response() :: {ok, correlation_id(), messages()} |
                    {incomplete, corr_id(), messages(), state()} |
                    error().
```

FAULT TOLERANCE

Backups!

- **ETS**, until receiving confirmation.
- **Disk**, if the ETS becomes too big.

ETS tables are also **supervised** separately.



BROKER CHANGES

The same features that provide fault tolerance allow us to handle this changes gracefully

The broker connections hold no information

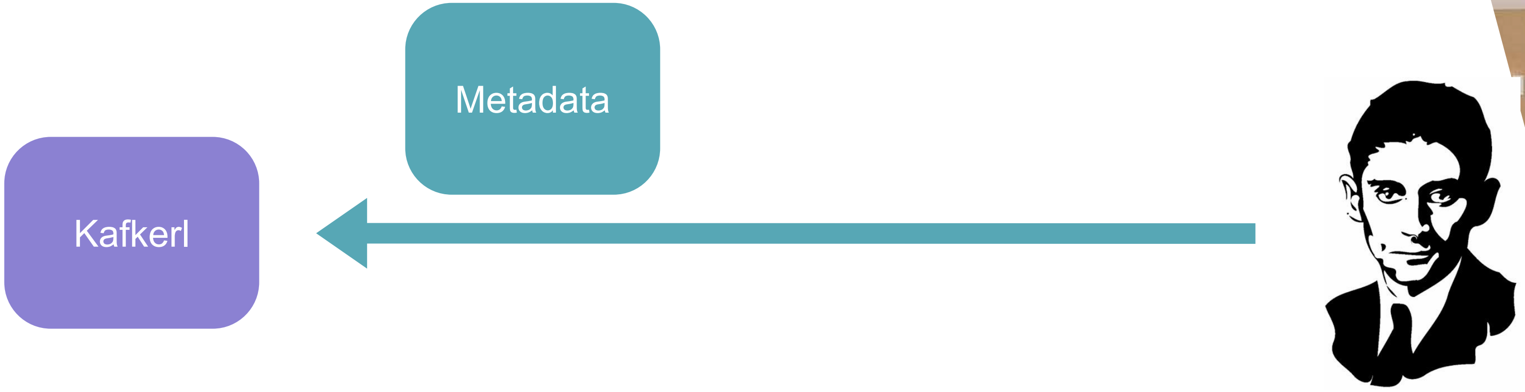
DON'T LET IT CRASH

Usually a good idea.

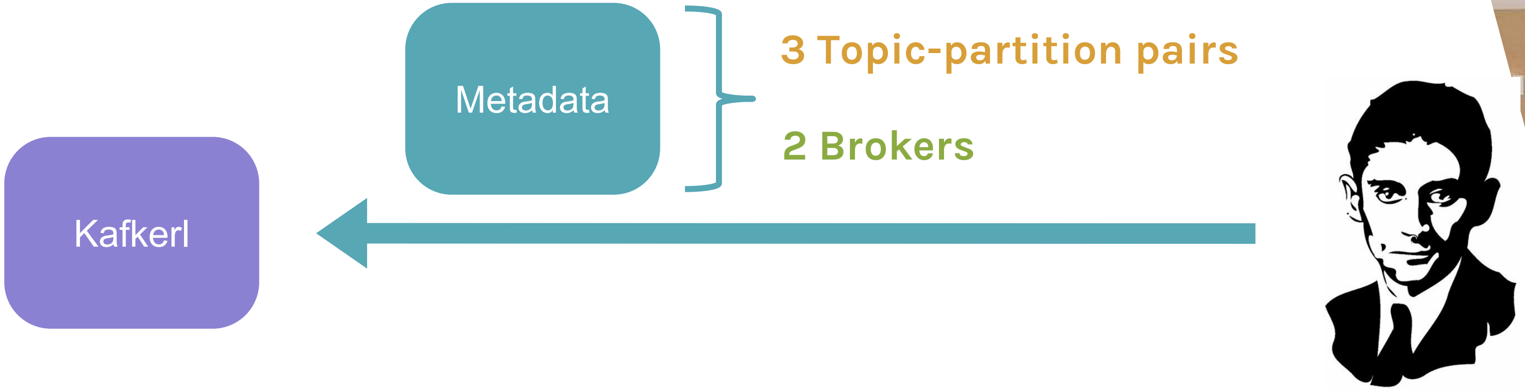
Not this time.

Using ETS tables for **concurrency**

Kafkerl starts by requesting the **metadata** to the **Kafka server**



Kafkerl starts by requesting the **metadata** to the **Kafka server**



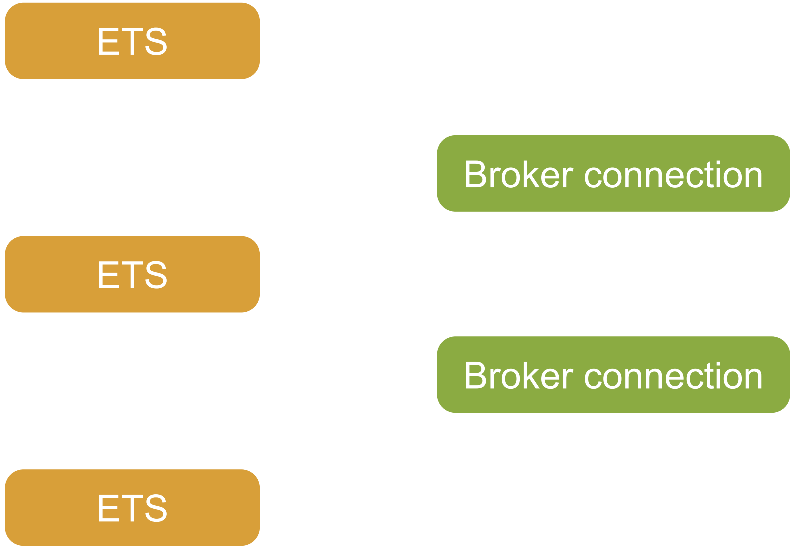
In this case, the **metadata** has 2 **brokers** and 3 **topics-partitions**

ETS

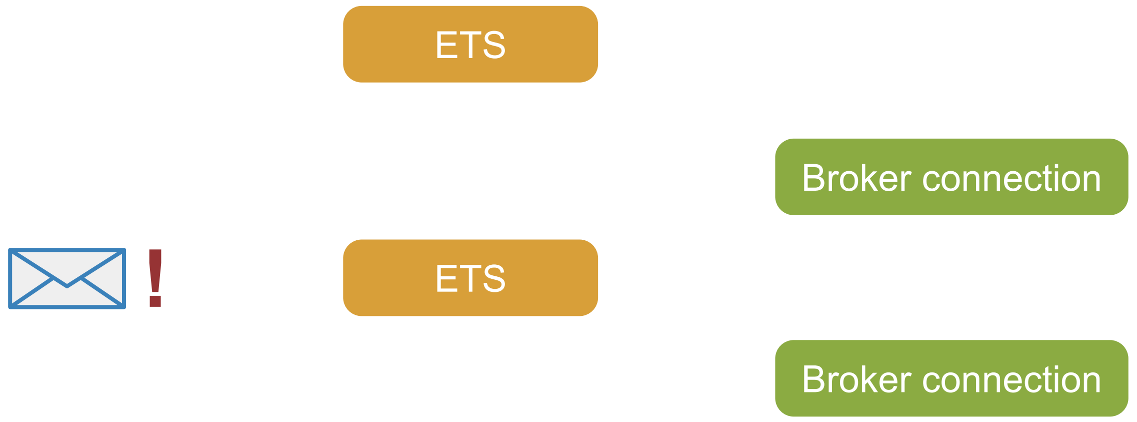
ETS

ETS

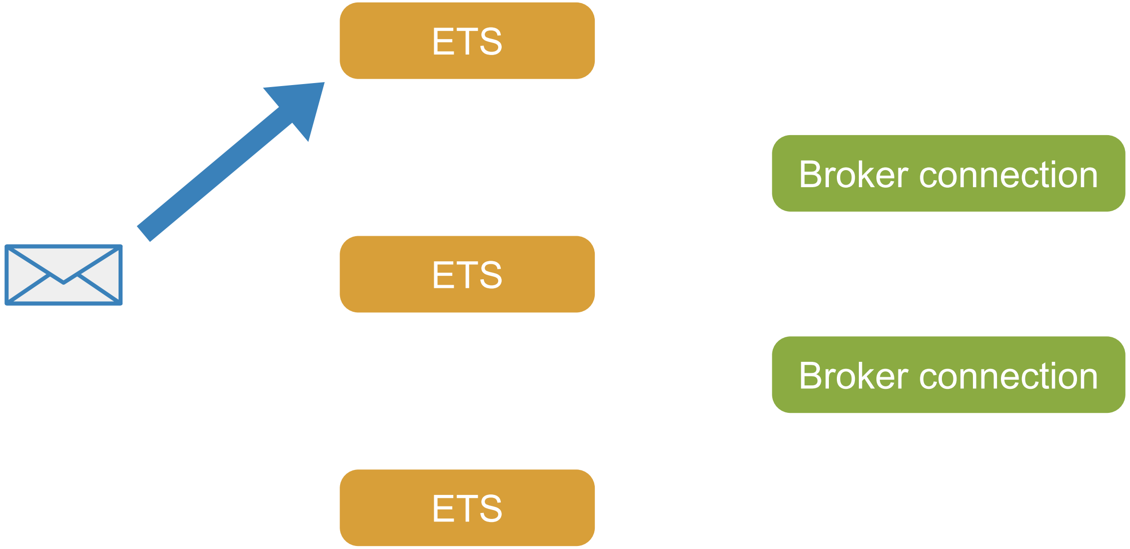
Then, we create an **ETS table** per **topic-partition** pair on the **Kafka server**



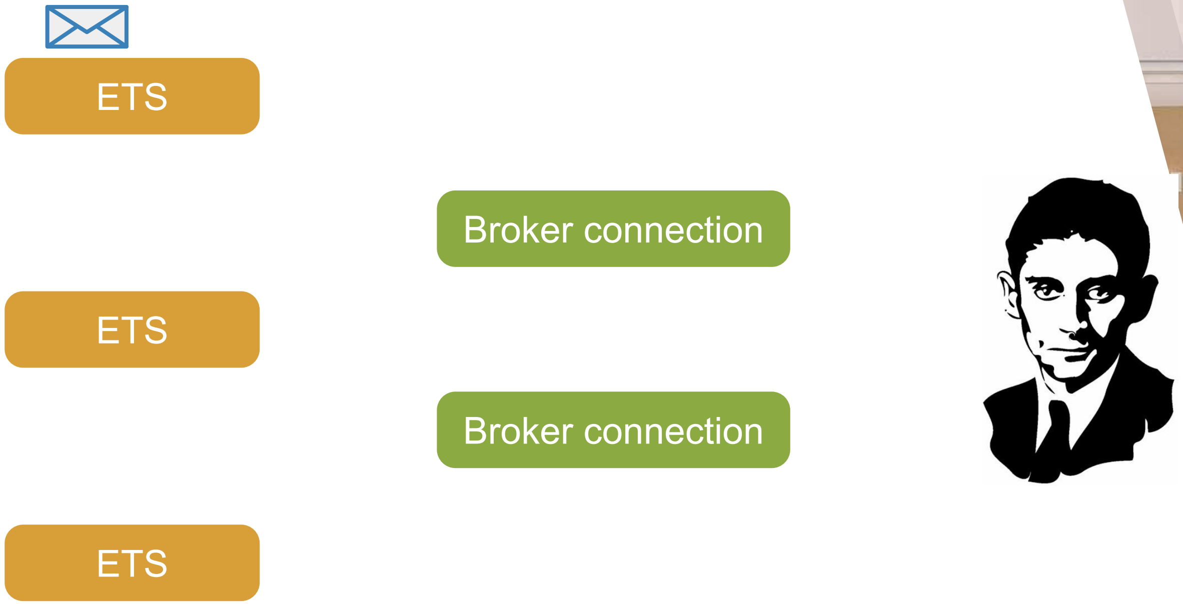
And one **broker connection** per **Kafka broker**



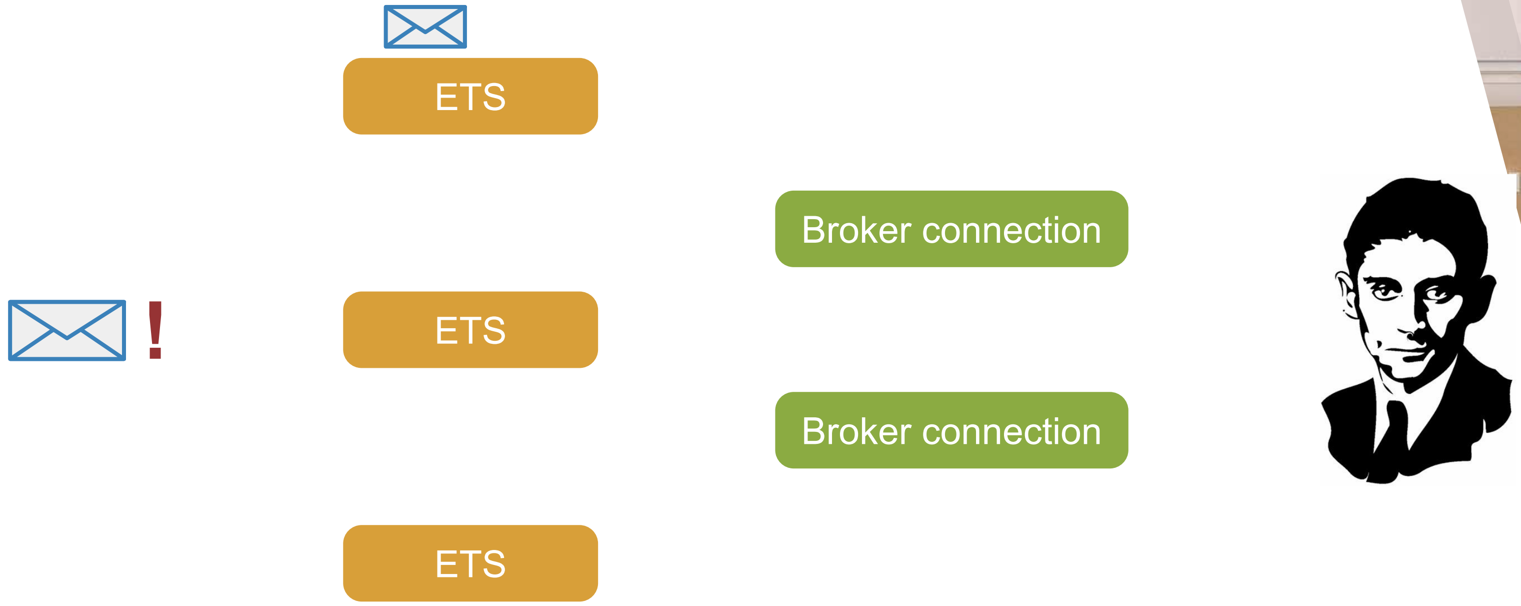
A **message** arrives from an **erlang process**!



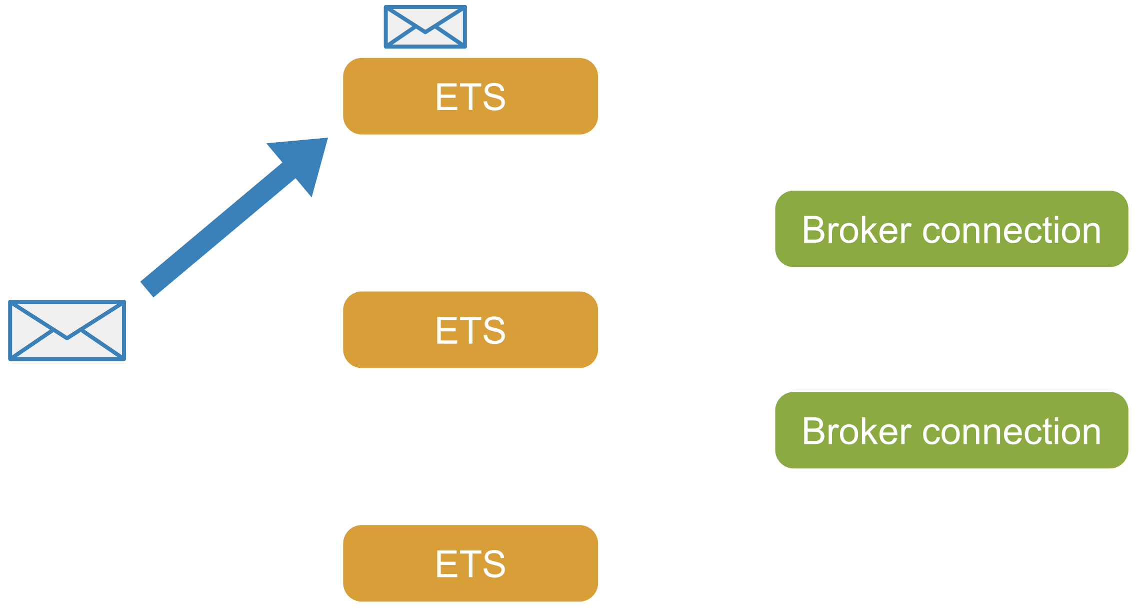
The **message** is routed to the right **ETS**



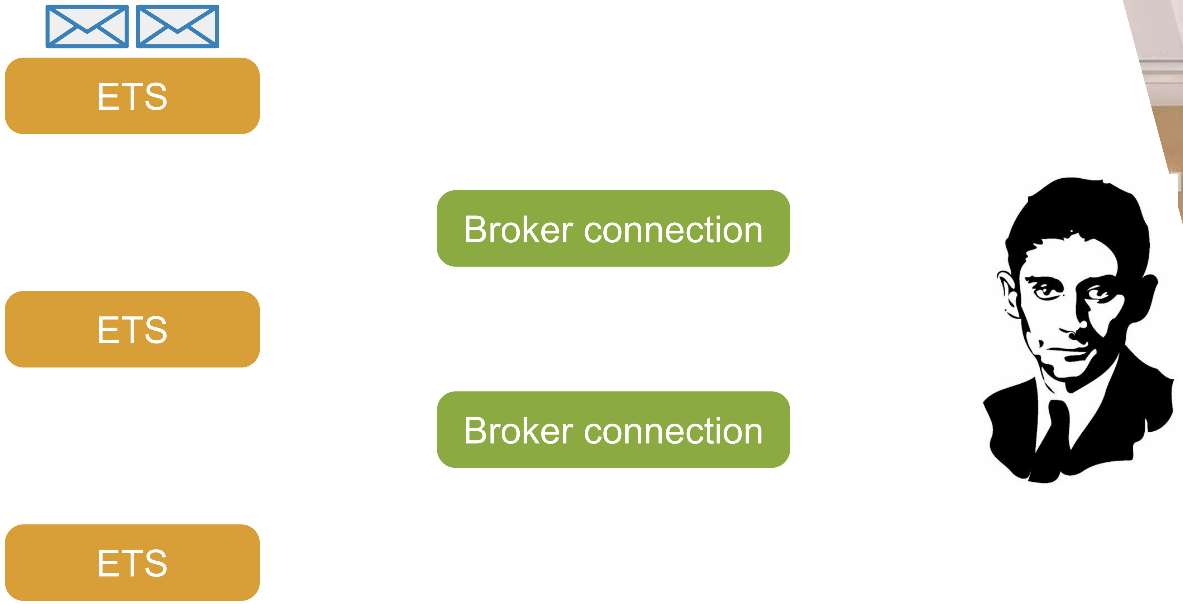
The **message** is routed to the right **ETS**



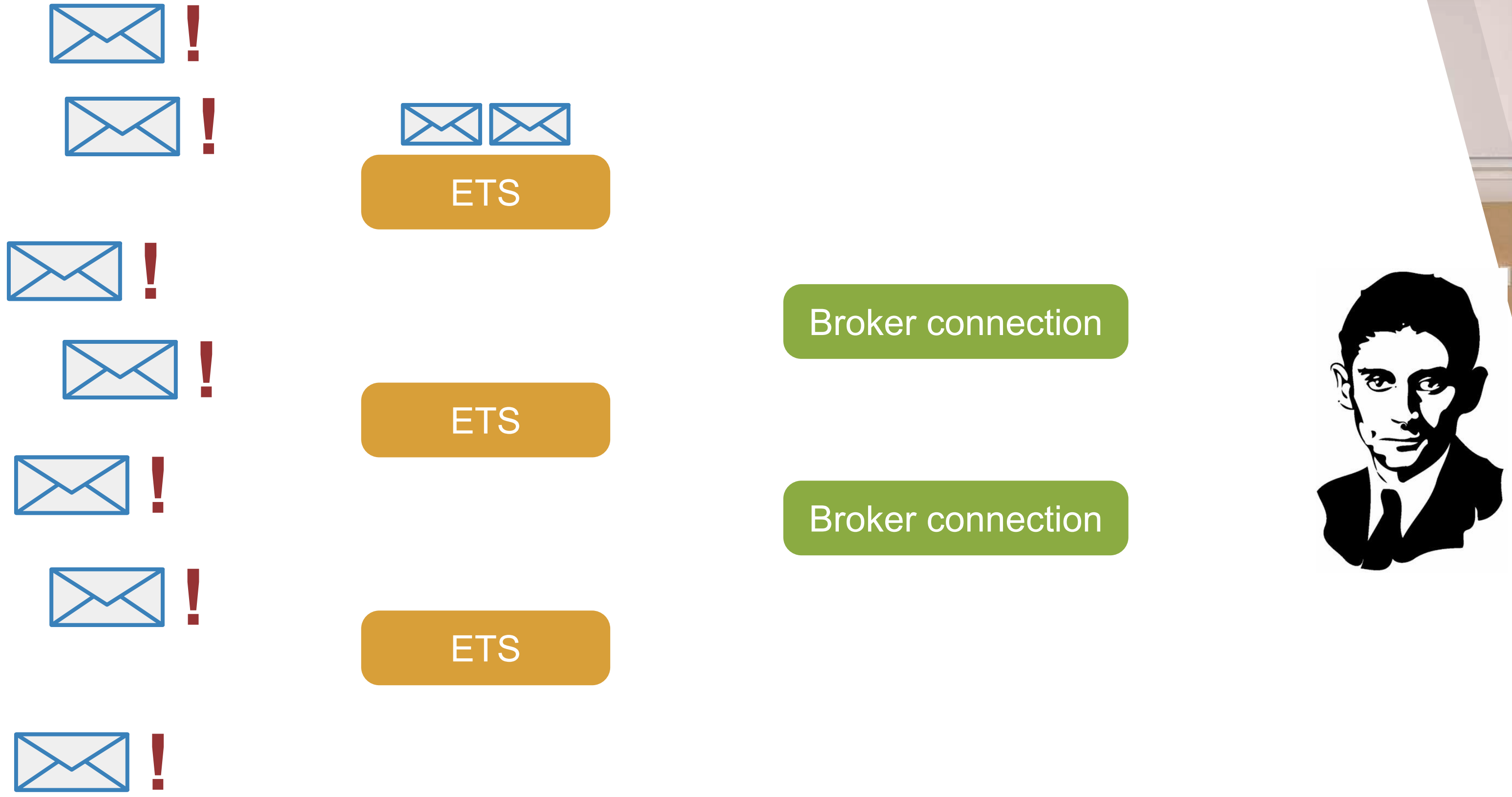
Another **message** arrives!



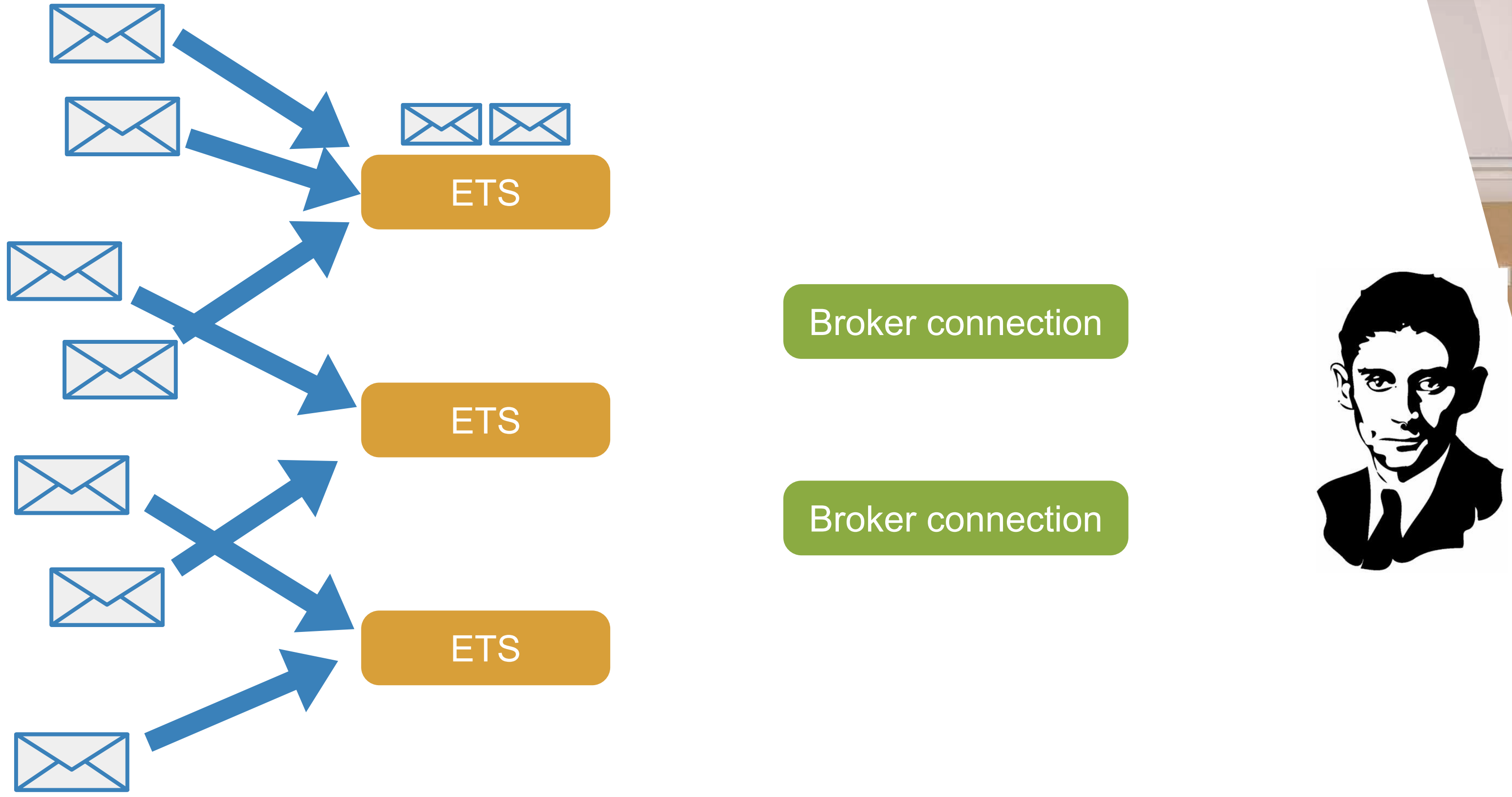
And again, the **message** is sent to the right **ETS** www.erlang-solutions.com



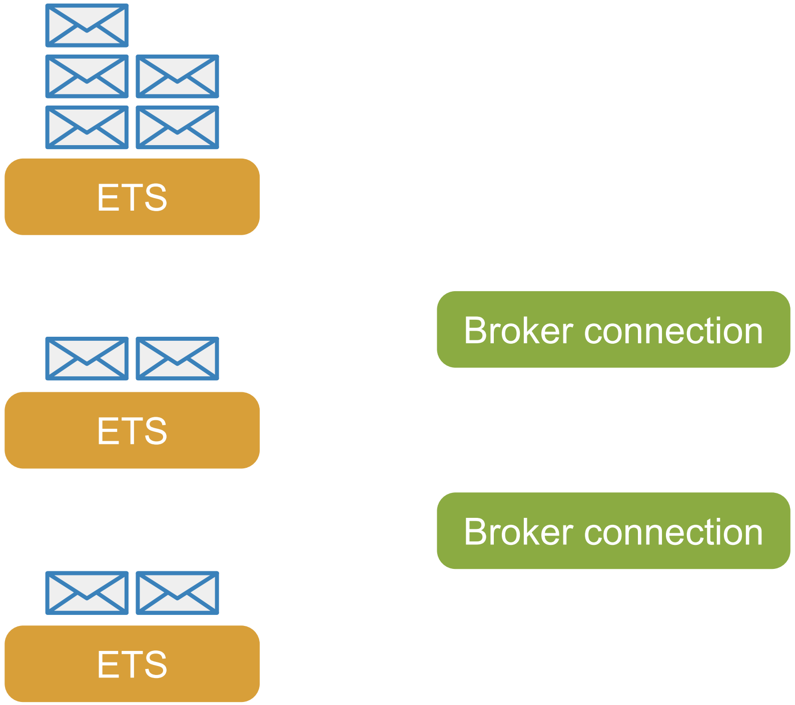
Remember that the **message** was written by the **process that created it**



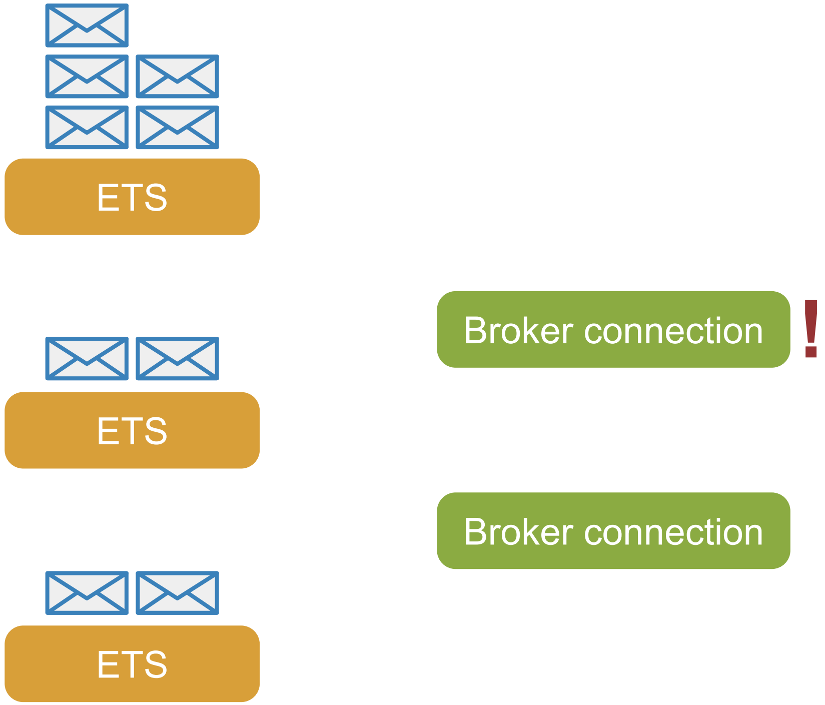
If **multiple messages** arrive



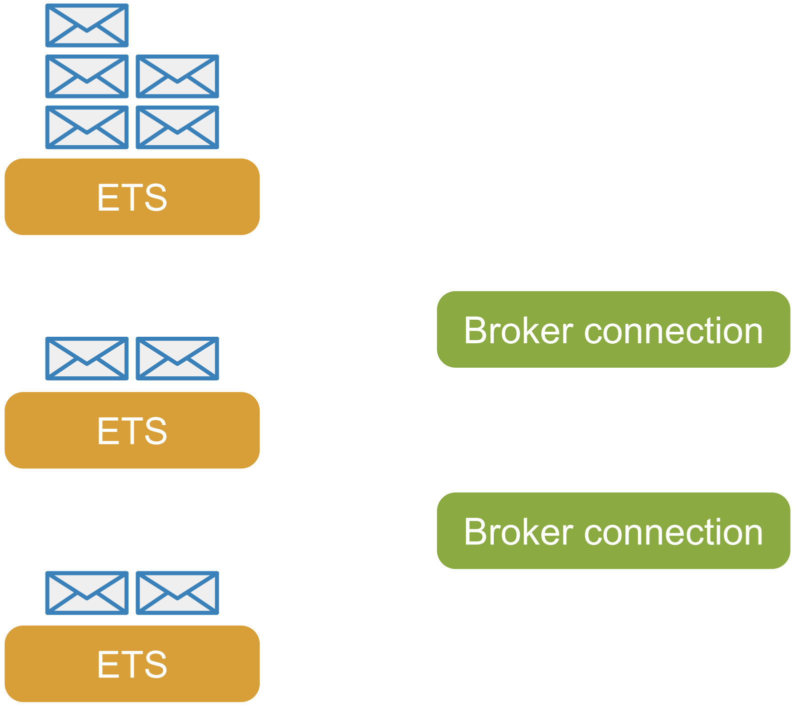
They are concurrently written to the **ETS tables**



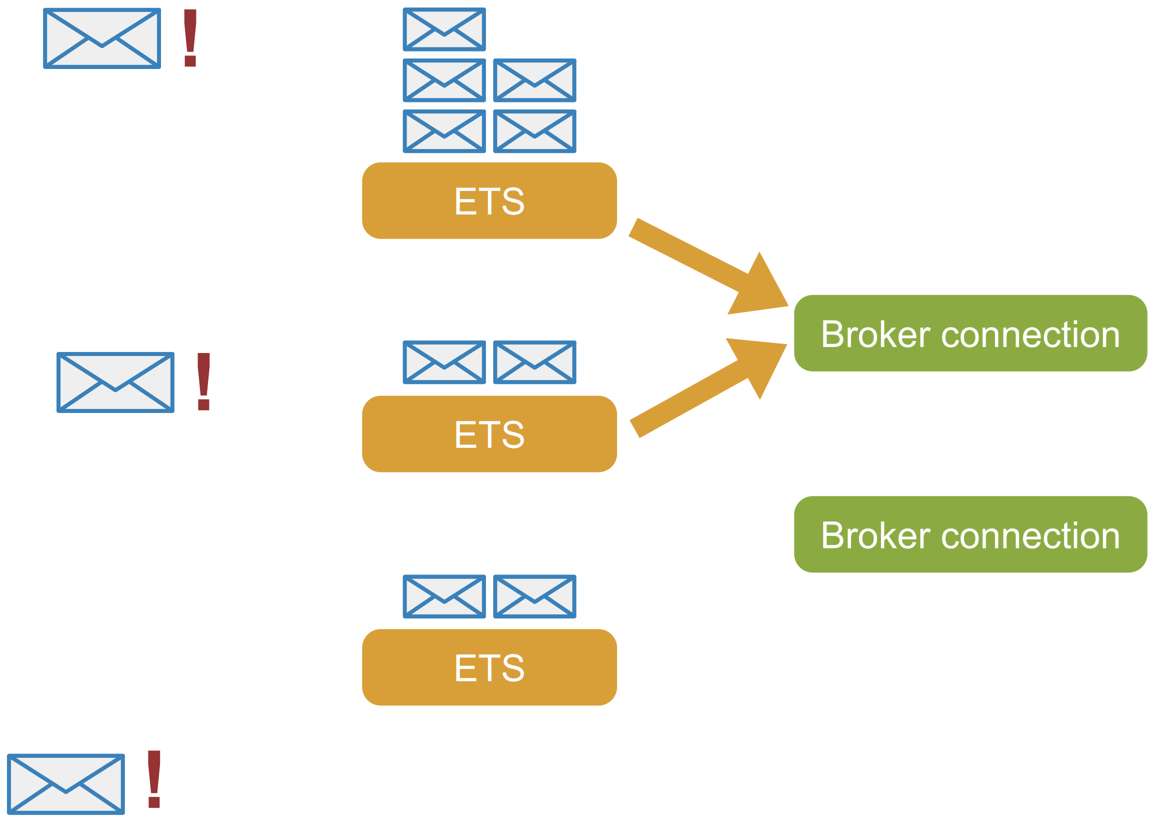
And the whole operation puts **no pressure** in the system, just the **tables**



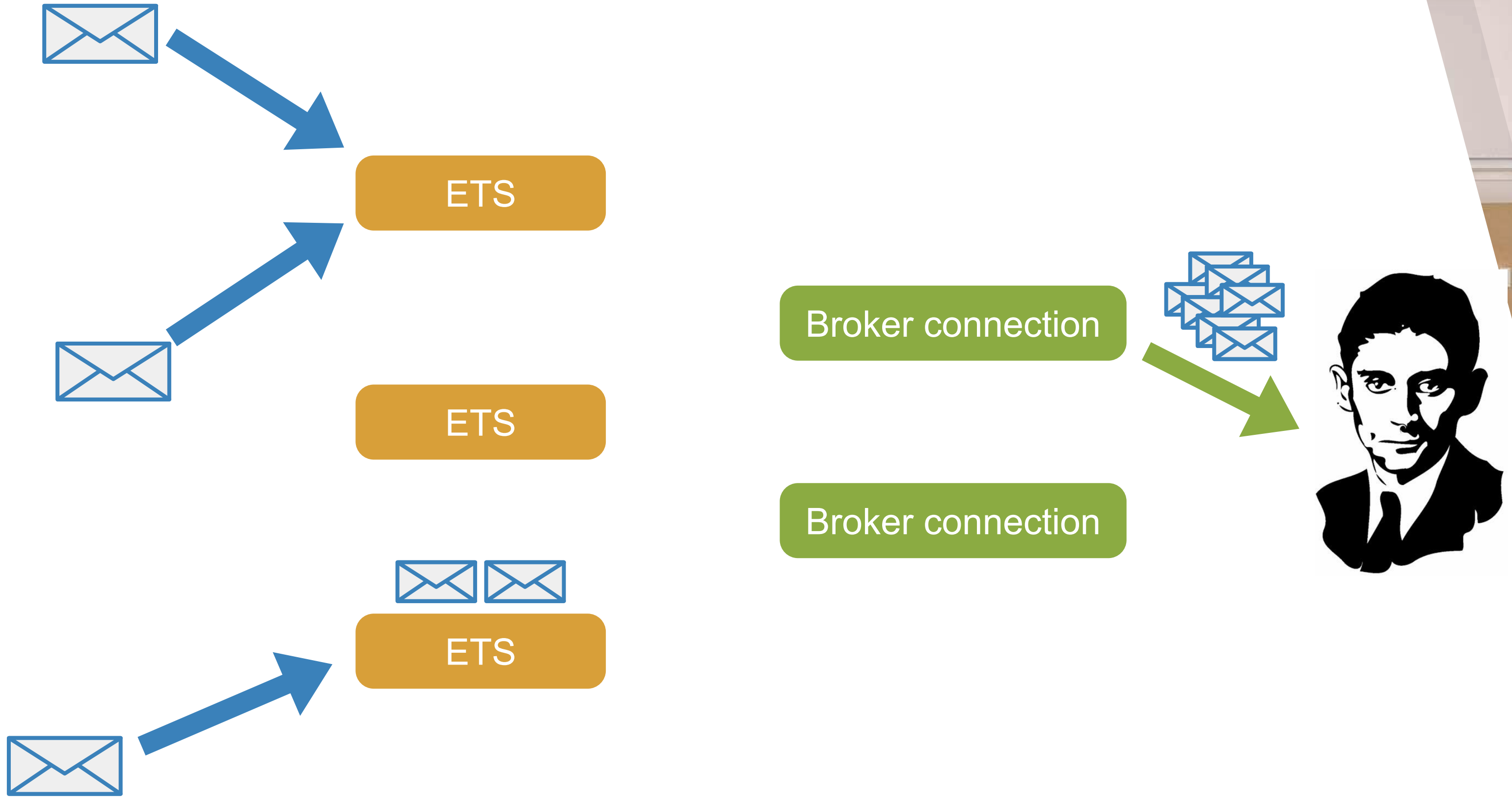
Now, a **broker** decides to publish the messages



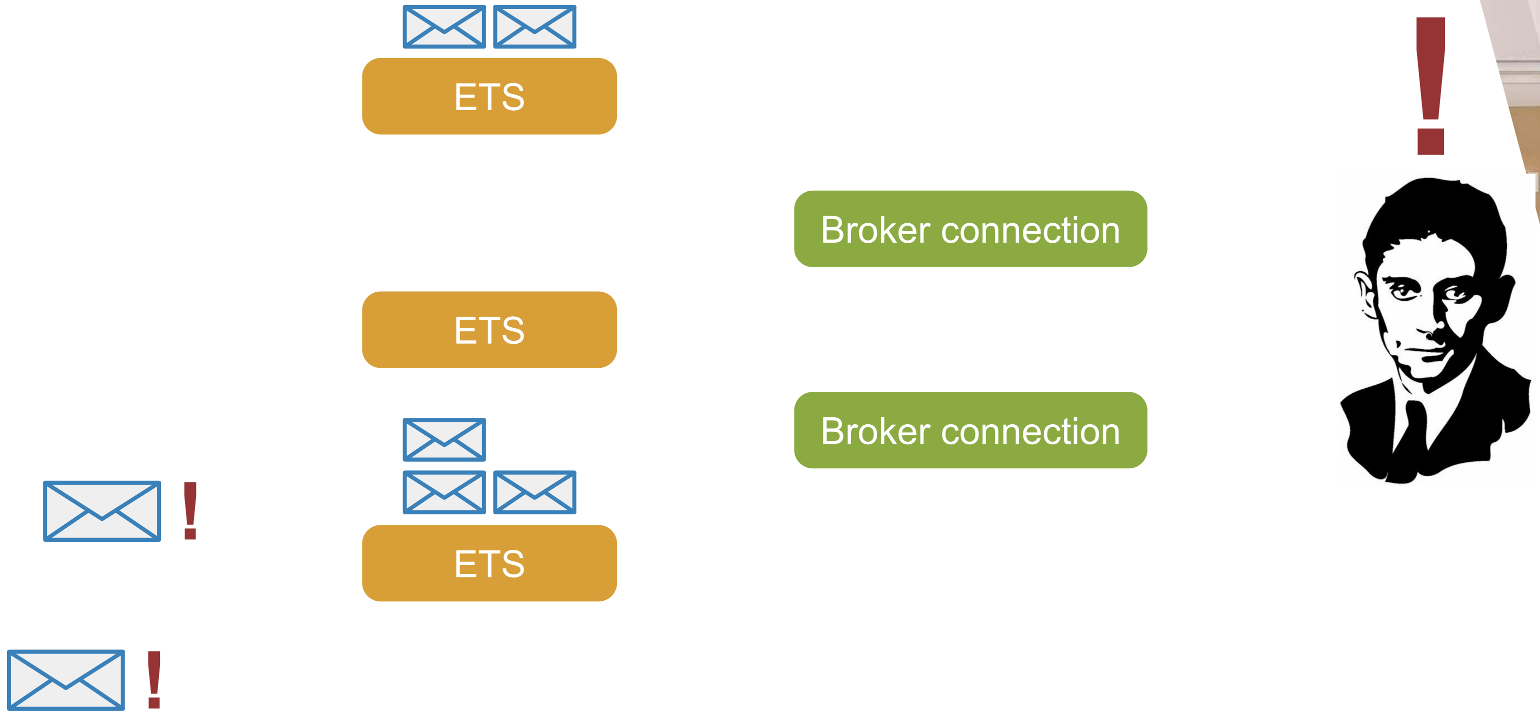
The **brokers** are not tied to the **ETS tables**



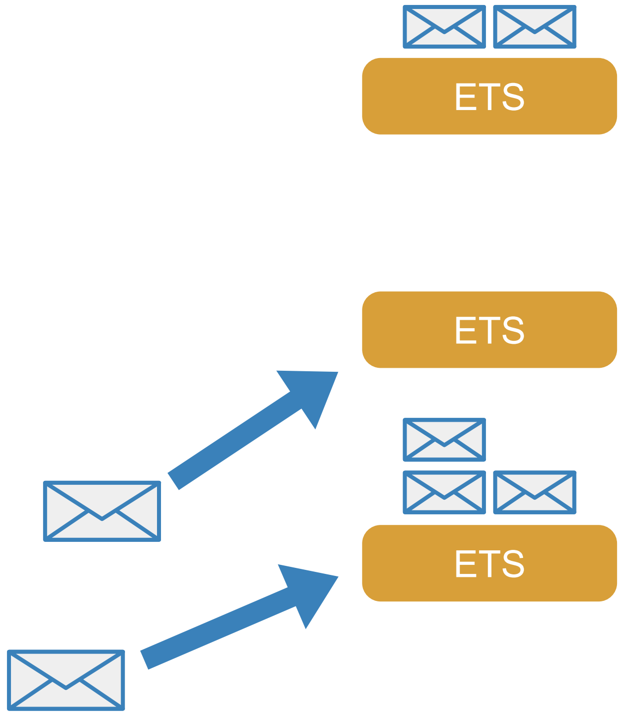
This **broker** reads all **messages** from 2 different **tables**



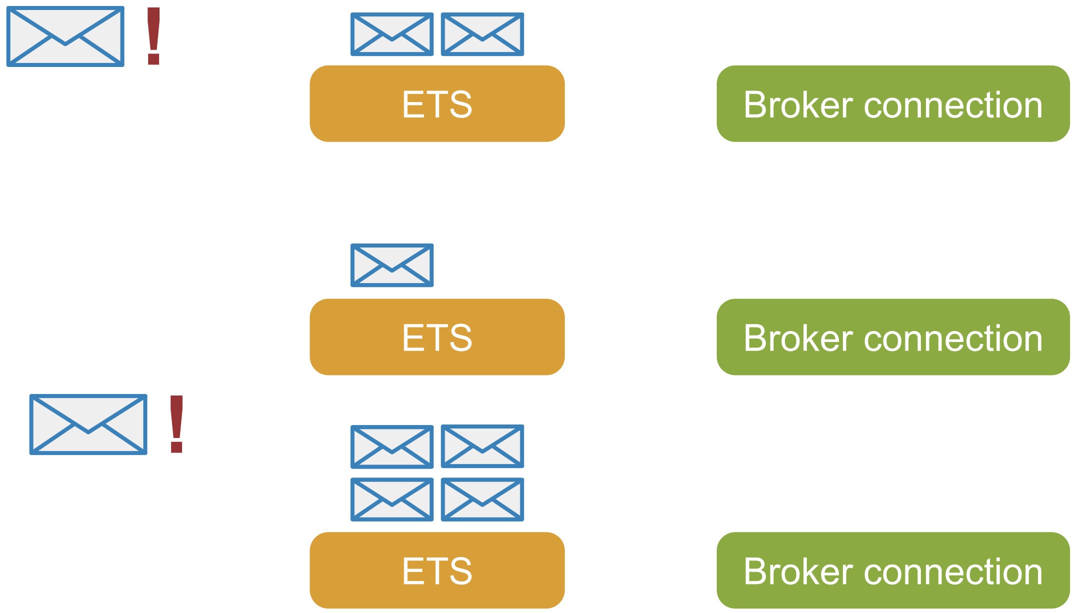
The **broker** builds and sends the binary, but the system works **as before**



Now, we receive a message about a **leadership change**



We remove both **brokers**



And **seamlessly** rebuild the **brokers** according to the new **leadership**

IN SUMMARY

All messages are stored in ETS tables, so we can take advantage of this built-in **BIF** to allow **simultaneous writes/reads**.

Messages are **not serialized** so big bursts of messages will not affect performance.

No error in logic can crash any **single process** holding the unsent messages.

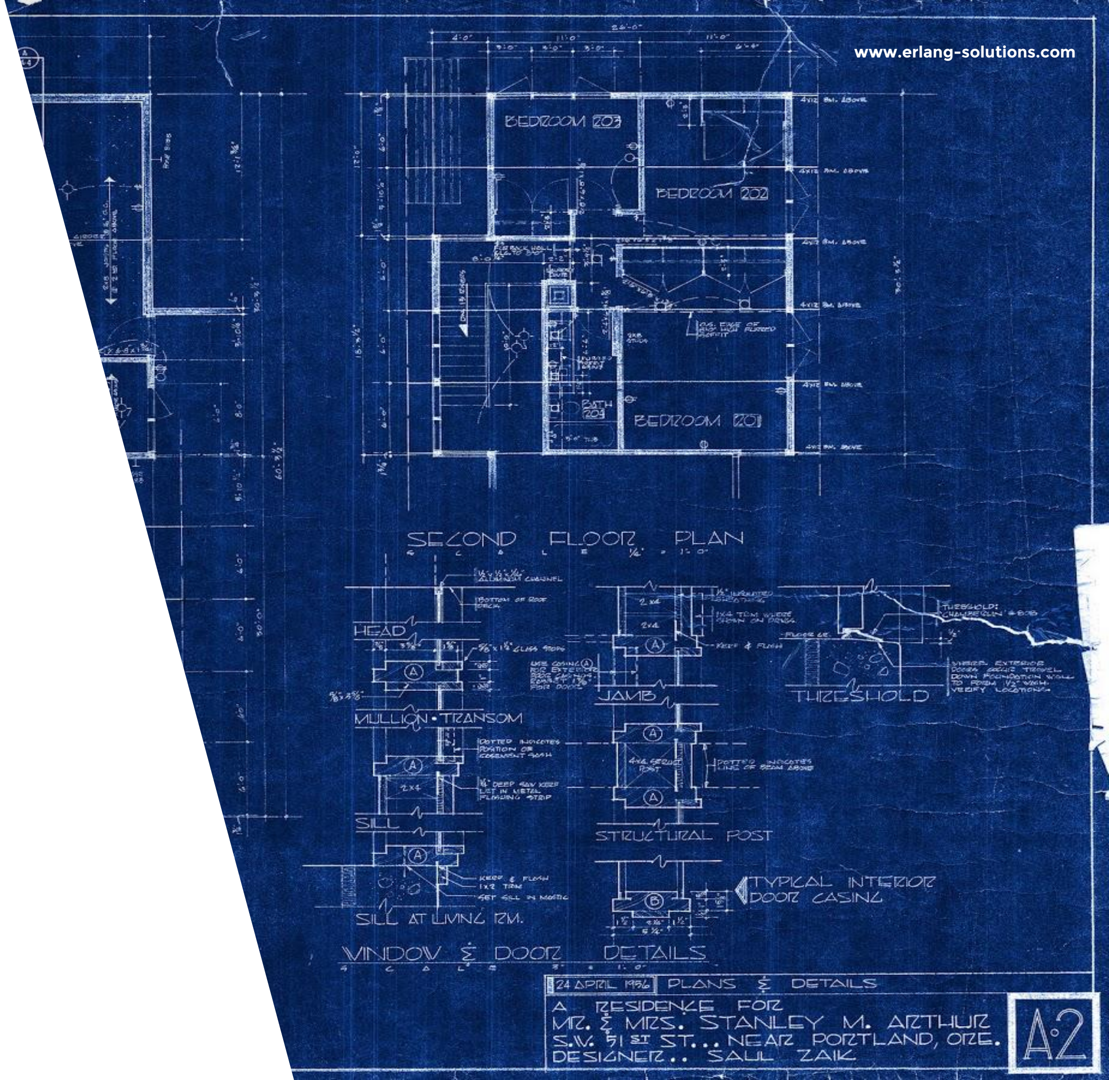
AND A BETTER SUMMARY

No downtime

No missing messages

No special cases

No errors


BEDROOM 203
BEDROOM 202
BEDROOM 201
BATH 204
SECOND FLOOR PLAN
HEAD
JAMB
THRESHOLD
MULLION • TRANSOM
SILL
STRUCTURAL POST
SILL AT LIVING RM.
TYPICAL INTERIOR DOOR CASING
WINDOW & DOOR DETAILS
24 APRIL 1956 PLANS & DETAILS
A RESIDENCE FOR
MR. & MRS. STANLEY M. ARTHUR
S.W. 51 ST ST... NEAR PORTLAND, ORE.
DESIGNER.. SAUL ZAIK
A2

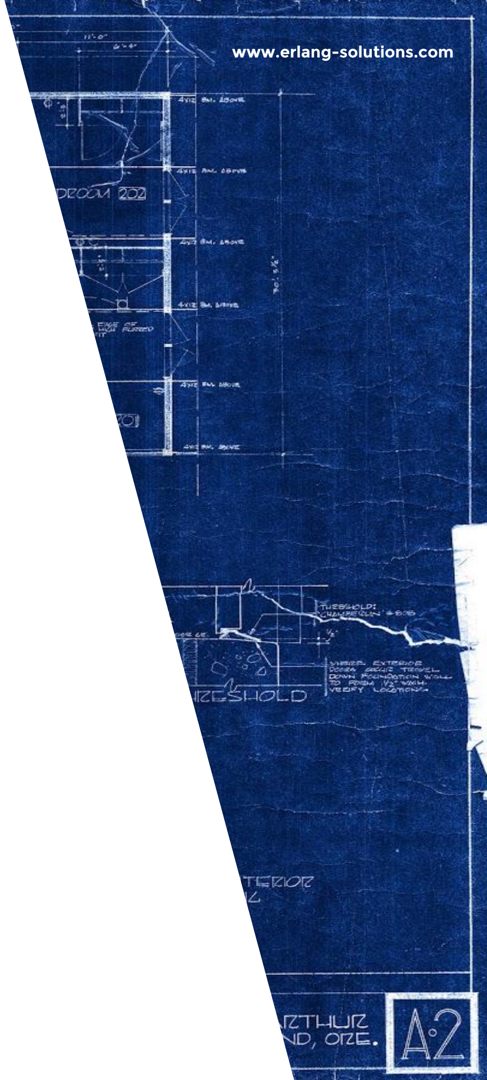
Good design

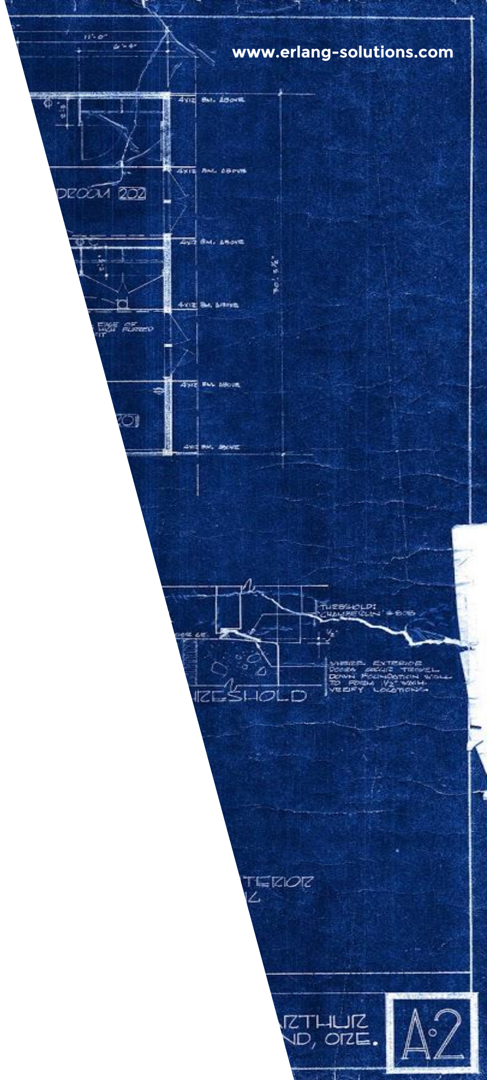
REINVENTING THE WHEEL

We chose a service already (like **Kafka**), why would you **reinvent the wheel** when designing the API?

Be consistent when **naming methods**.

Prefer **simple datatypes** and avoid **complex structures**.





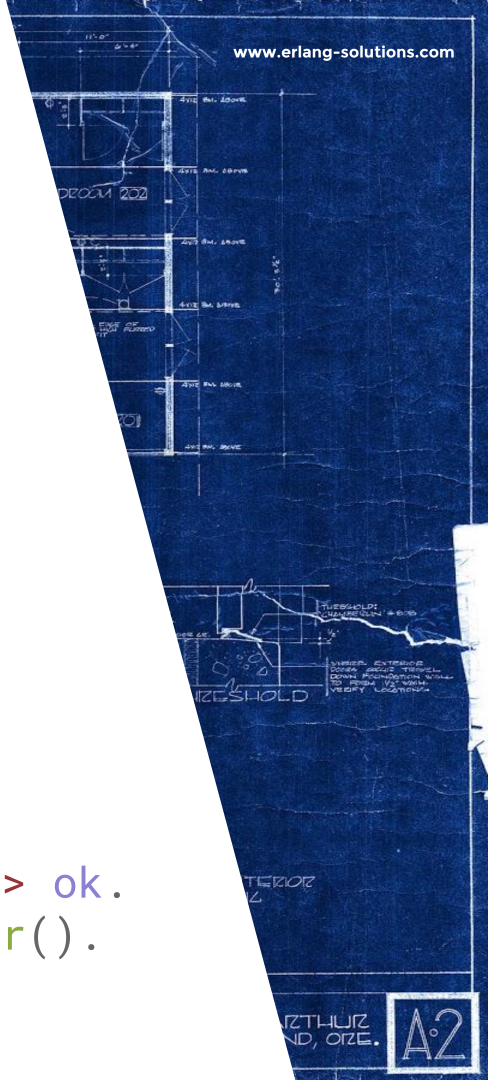
MAKING IT EASIER

If you make the interface **similar to the service**, **anyone** can pick it up!

If there's one thing you can count on is that all your **users should be familiar** with it.

For Example:

```
-export([produce/3, produce/4, produce/5,
         consume/2, stop_consuming/2,
         request_metadata/0]).
```



GOOD **API** DESIGN

Let the API accept very **strict datatypes** (ie not being liberal in what you accept)

Any programming **errors** will be caught quickly and it also serves as **documentation**.

No surprises, **no** edge cases.

For Example:

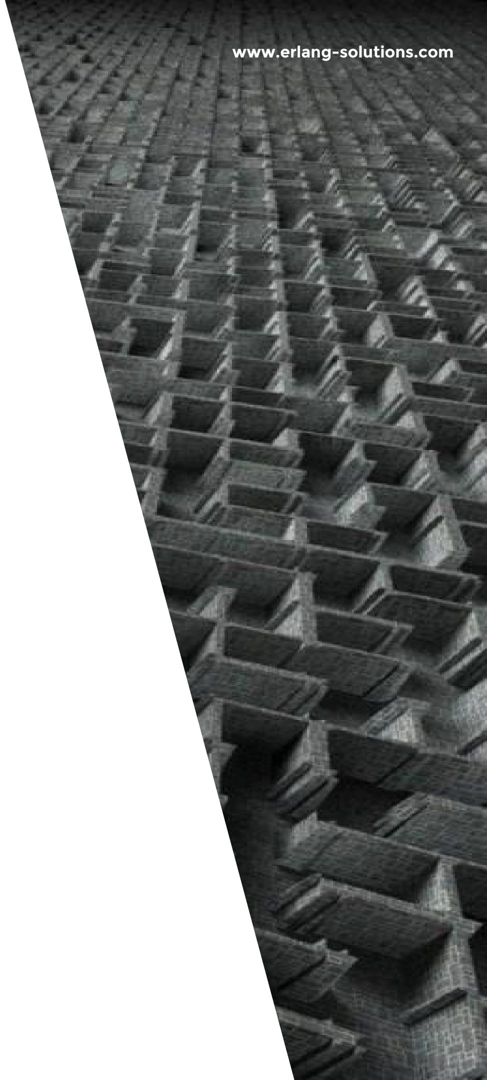
```
-spec produce(topic(), partition(), payload()) -> ok.
-spec consume(topic(), partition()) -> ok | error().
-spec request_metadata() -> ok.
```

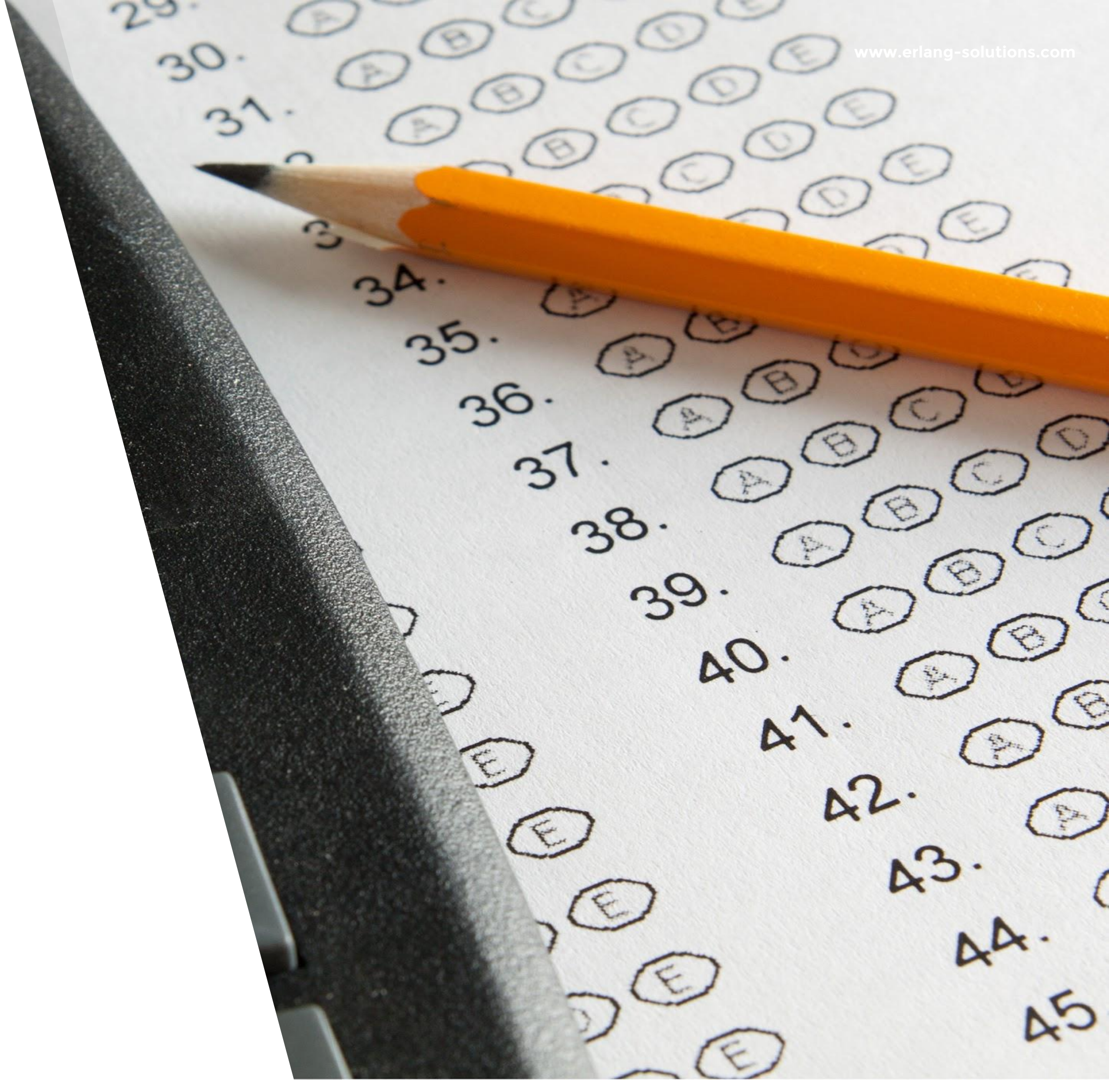
SIMPLIFYING

Easy to understand API

Put everything into a **single module**.

Hide the complexity from the developer.





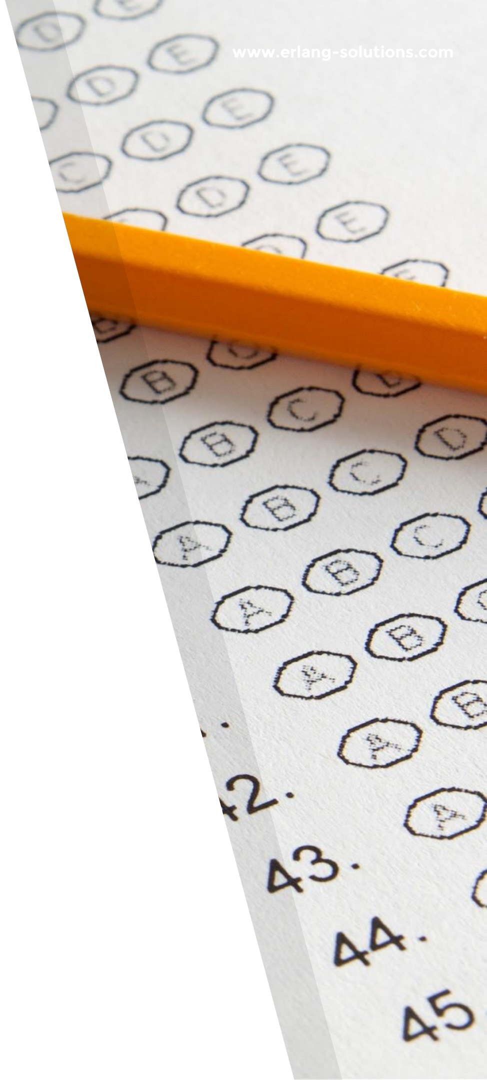
Testing the untesteable

dialyzer

Should always be the **first step**.

Prevents **basic errors**.

Changes in plans (and we had many) can leave traces, **dialyzer** tracks them down.

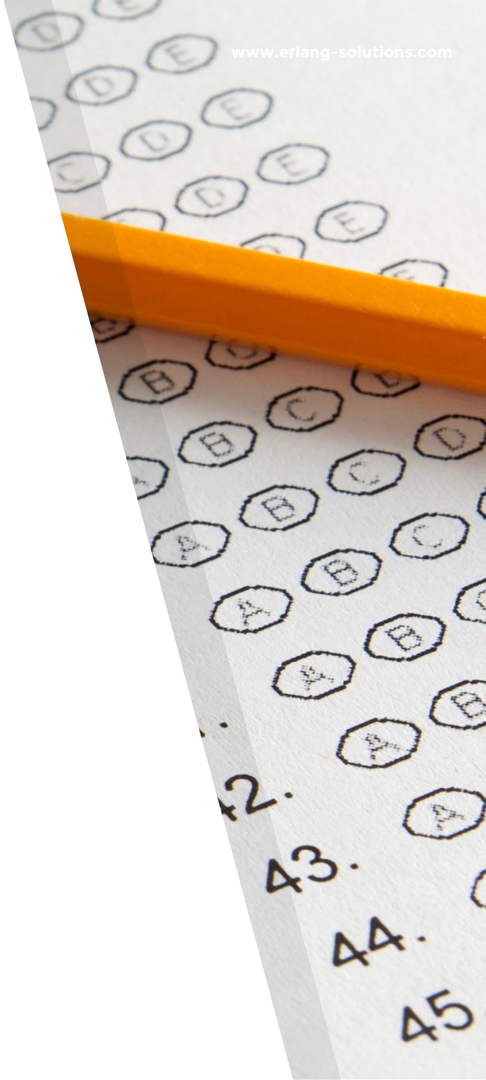


UNIT TESTING

Even without integration tests, we should make sure different parts of the system work properly

Things that can be tested this way:

- Protocol
- ETS tables
- The basic functionality

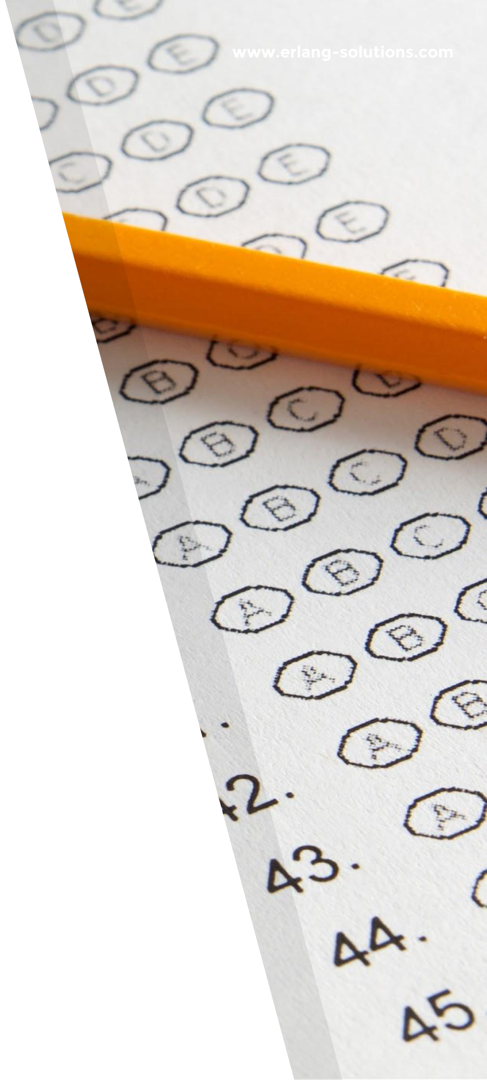


PropEr

Great way to test libraries since we have strict contracts that we need to make sure we are obeying?

PropEr

A QuickCheck-Inspired Property-Based Testing Tool for Erlang



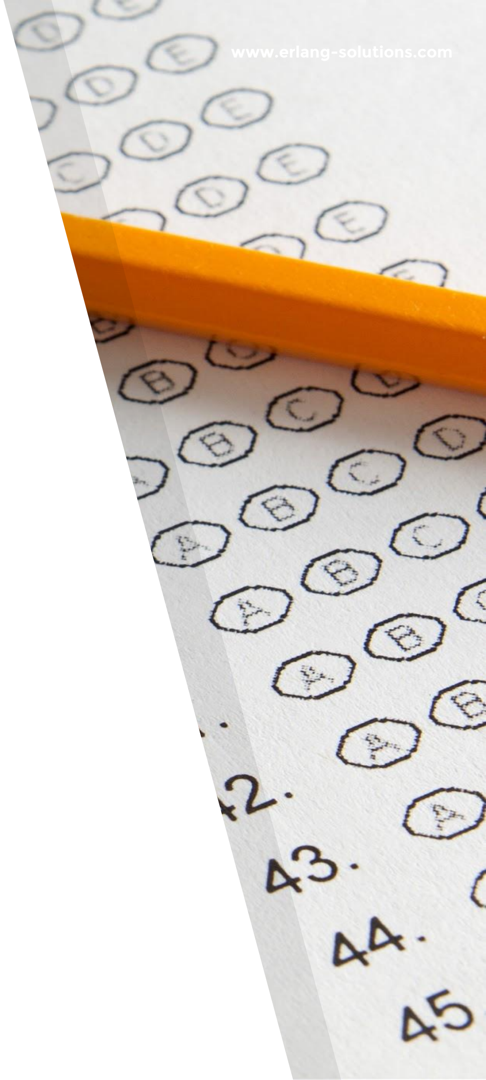
g-solutions.com

IT'S ALIVE!

Blackbox testing is not enough.

Use the **staging environment** as early as possible.

Errors will be found while debugging other parts of the system.



So, how did it go?

8 Months passed...

There was a downtime...

Whatever, just find the logs on disk!

OH BOY!

We had no permission to write!

But it did last 8 months!

THANK YOU!

Any questions?

hernan@inakanetworks.com
www.erlang-solutions.com www.inaka.net
@inaka