Návrh systémů IoT

4. Zasílání zpráv v prostředí IoT, message brokery.

Stanislav Vítek
Katedra radioelektroniky
České vysoké učení technické v Praze

Obsah přednášky

1. Protokoly pro zasílání zpráv
2. Zprostředkovatelé komunikace - brokery

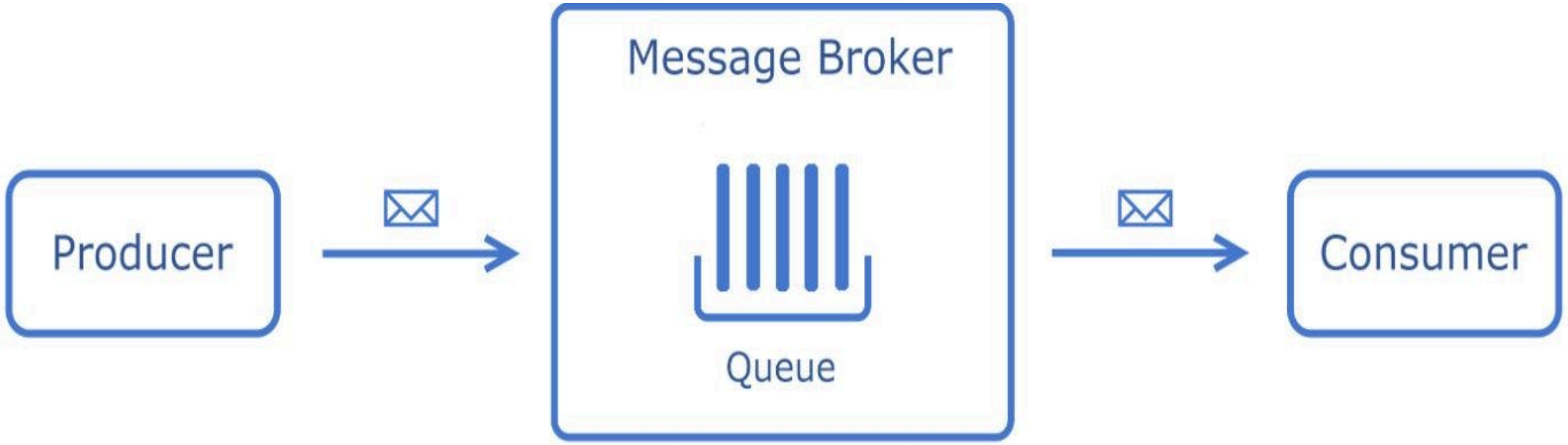
Protokoly vyšších vrstev vs. IoT

- Použití HTTP protokolu je pro IoT problematické
  - Omezené možnosti IoT prostředí (zejména konektivita)
  - HTTP hlavička reprezentuje nejméně 71B
- Problémy s firewally
  - Skrytý zdroj zprávy příchozí z NAT sítí
  - v IoT sítích ale potřebujeme vědět, kdo data posílá
- Dokumenty založenné na XML jsou příliš objemné

Role protokolů vyšších vrstev

1. Poskytování abstrakce "zprávy" (elementární jednotky dat), komunikace mezi koncovými body.
2. Poskytování primitiv pro datovou komunikaci/výměnu zpráv aplikacím vyšší vrstvy.
3. Implementace specifických síťových paradigmat.
4. Poskytování dodatečných mechanismů spolehlivosti nebo zabezpečení.
5. Někdy přizpůsobení již existujících (nikoliv nativně M2M) řešení.

Zprostředkovatel komunikace - broker



K čemu broker slouží?

- **Asynchronní komunikace:** Umožňuje různým částem systému komunikovat bez nutnosti okamžité reakce, což vede k efektivnějšímu využívání zdrojů.
- **Oddělení služeb:** Umožňuje nezávislý provoz služeb, což snižuje složitost systému a zvyšuje jeho udržovatelnost a škálovatelnost.
- **Vyrovnávání zátěže:** Rozděluje zprávy rovnoměrně mezi různé služby nebo pracovníky, čímž pomáhá řídit pracovní zátěž a zlepšuje výkon systému.
- **Zachování pořadí:** Některé fronty zpráv mohou zajistit, aby byly zprávy zpracovány v pořadí, v jakém byly odeslány, což je pro specifické aplikace klíčové.

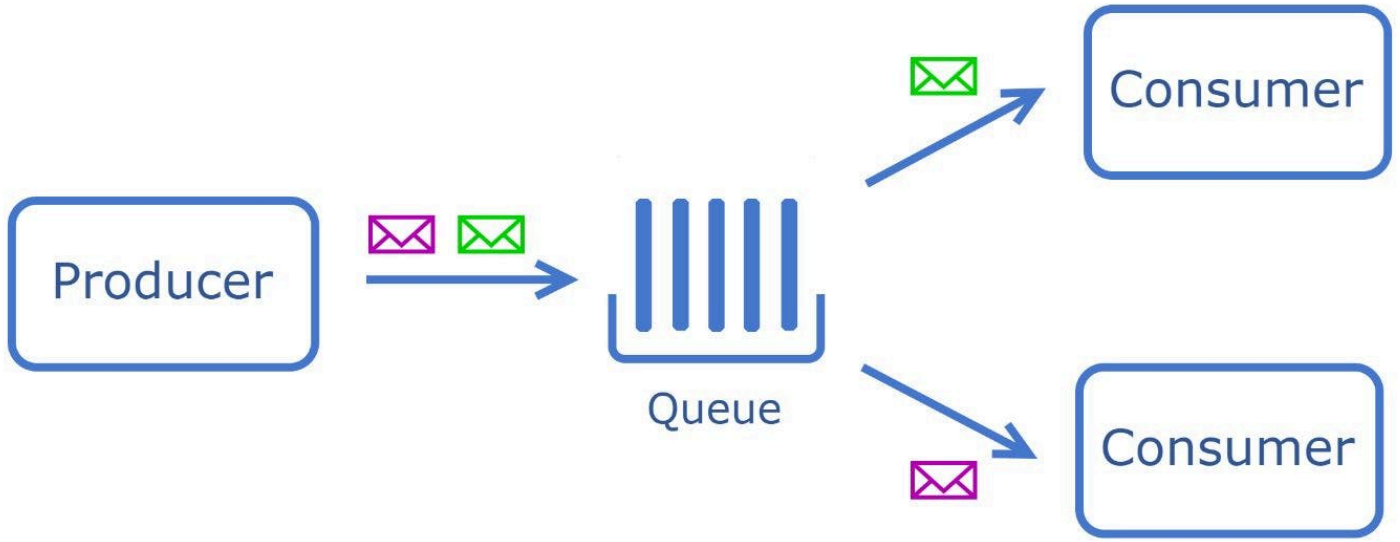
Vlastnosti brokerů

- **Škálovatelnost:** Usnadňuje snadné škálování aplikací přidáním dalších konzumentů nebo prostředků pro zpracování zvýšeného toku zpráv.
- **Omezování rychlosti a škrcení:** Řídí rychlost zpracování zpráv, což je důležité pro správu zdrojů a prevenci přetížení systému.
- **Schopnost rozfázování:** Fronty zpráv často obsahují mechanismus fan-out, který umožňuje doručit jednu zprávu více konzumentům nebo službám současně.
- **Perzistence dat:** Nabízí možnost ukládat zprávy na disk nebo do paměti, dokud nejsou úspěšně zpracovány, což zajišťuje, že data nebudou ztracena v případě selhání systému.
- **Filtrování a směrování zpráv:** Umožňuje směrování nebo filtrování zpráv na základě specifických kritérií nebo obsahu, což umožňuje cílenější a efektivnější zpracování.

Komunikační modely

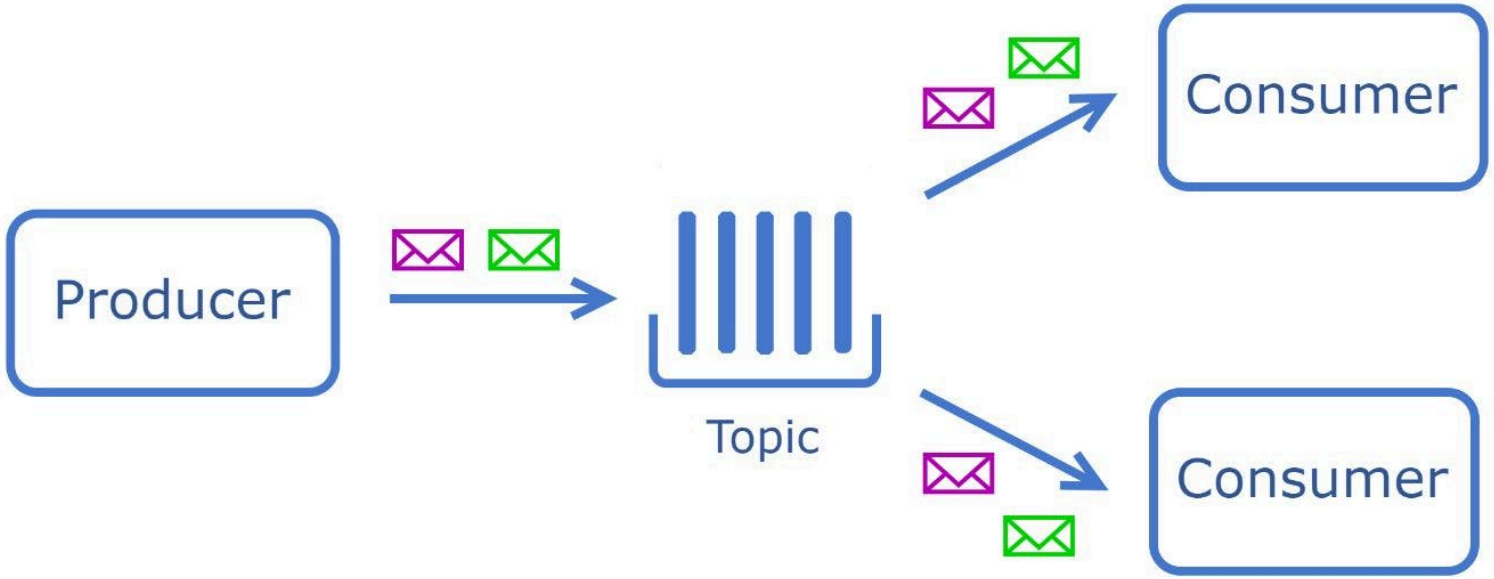
- Obecně lze říci, že existují dva hlavní modely komunikace, **Point-to-Point** a **Publish-Subscribe**.
- Nicméně, některé komunikační protokoly a brokery, které je podporují, využívají další komponentu nazvanou **Exchange** pro směrování.
    - V tomto případě jsou zprávy nejprve publikovány do výměny (Exchange) v rámci brokera.
    - Exchange, působící jako směrovací agent, přeposílá tyto zprávy do příslušné fronty podle svých směrovacích pravidel.

Point to point



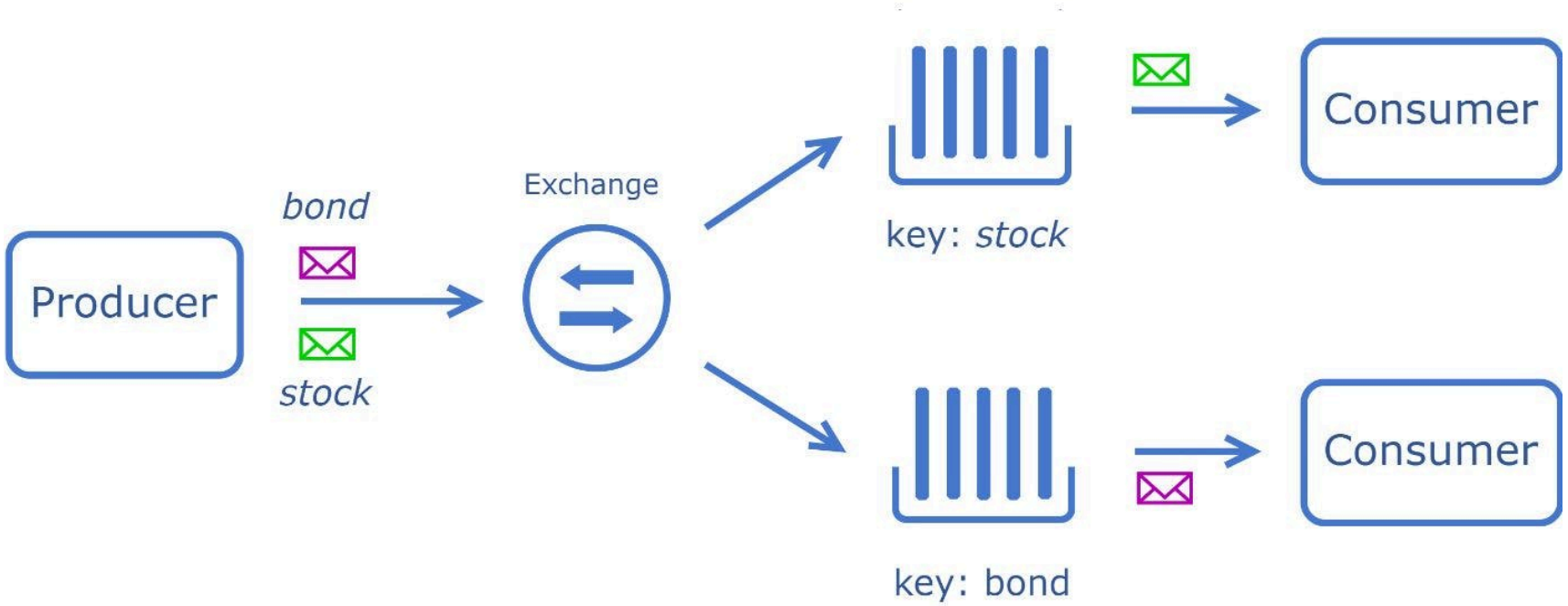
- Zprávy odeslané producentem jsou umístěny do fronty a jsou spotřebovány jedním spotřebitelem.
- To zajišťuje, že každá zpráva je zpracována pouze jednou jedním příjemcem.

Publish-Subscribe



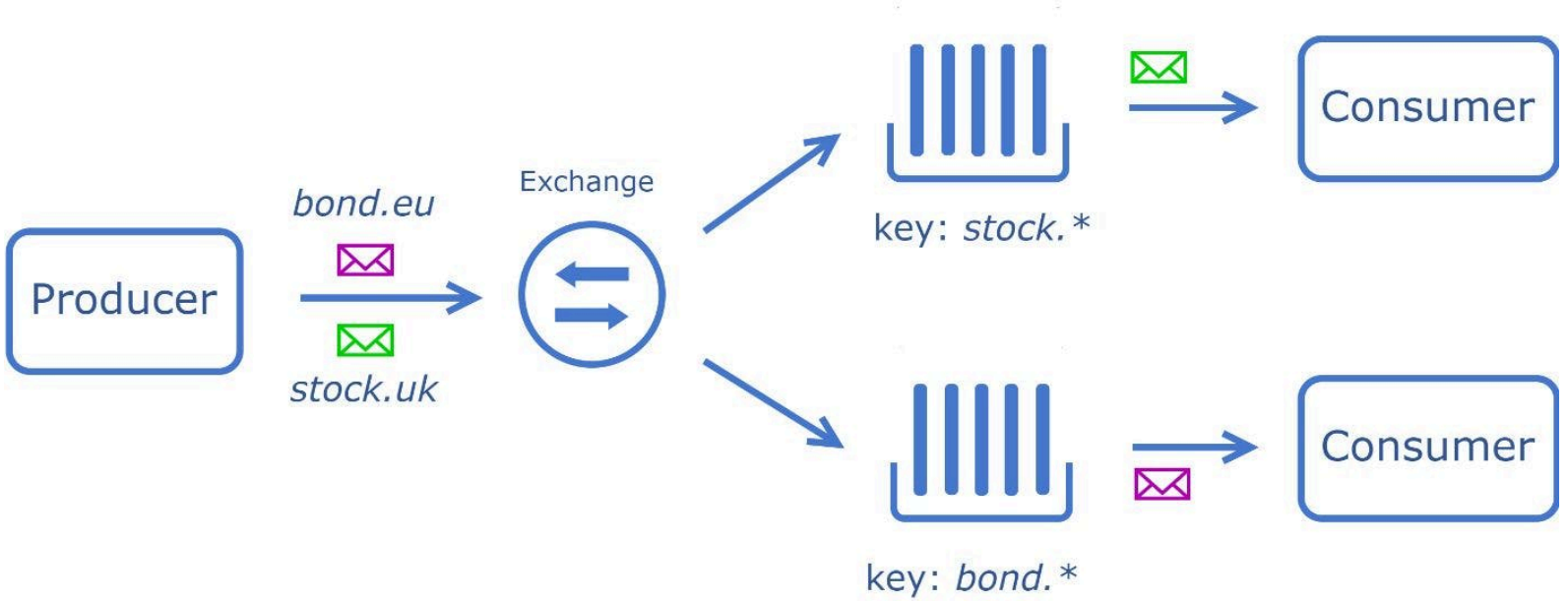
- Zprávy jsou publikovány do konkrétního tématu, nikoli do fronty.
- K jednomu tématu se může přihlásit více konzumentů a přijímat zprávy vysílané do tohoto tématu.

Direct Exchange



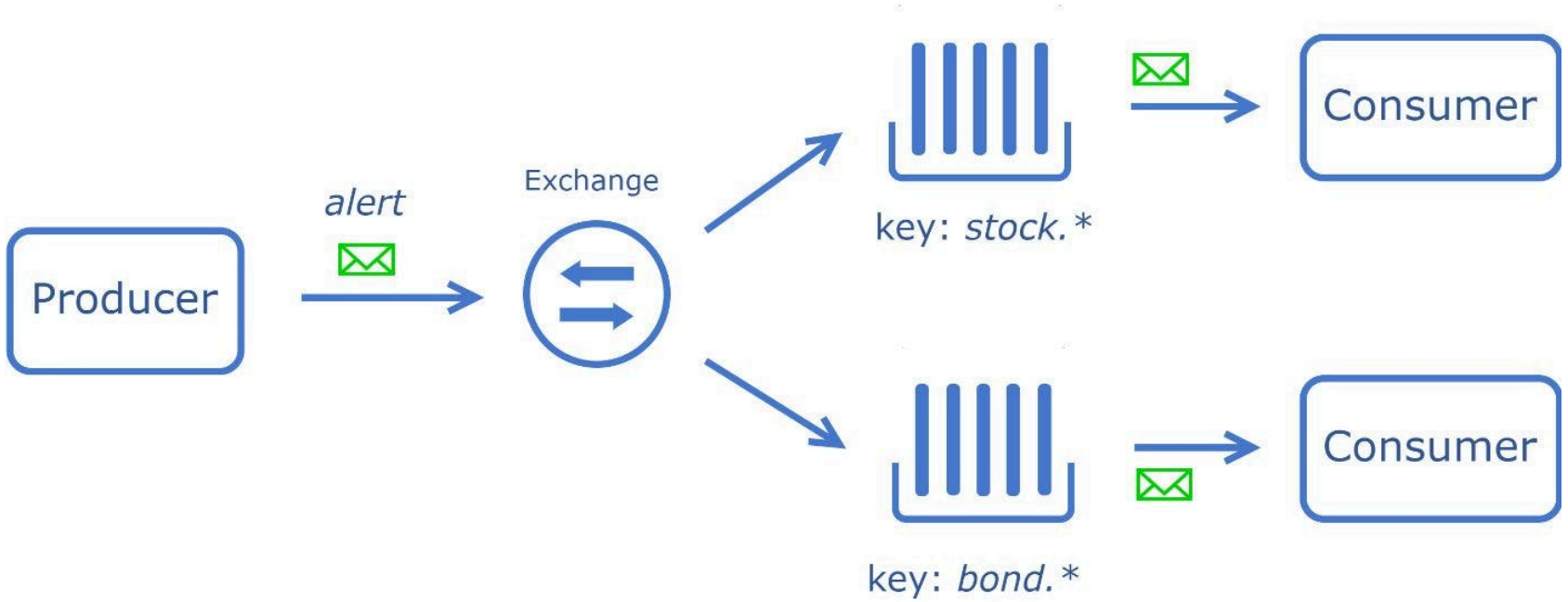
- Zpráva je směrována do front, jejichž vazební klíč se shoduje se směrovacím klíčem zprávy.

Topic Exchange



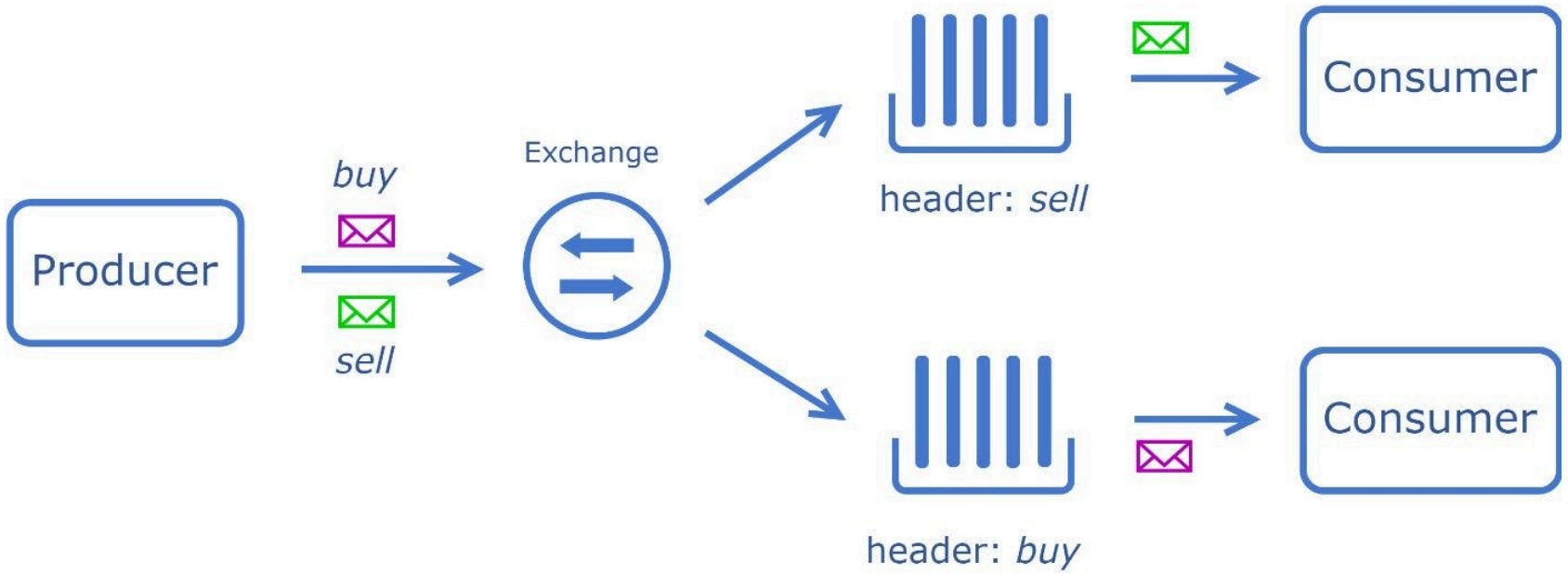
- Doručuje zprávy do jedné nebo více front na základě přiřazení témat.

Fanout Exchange



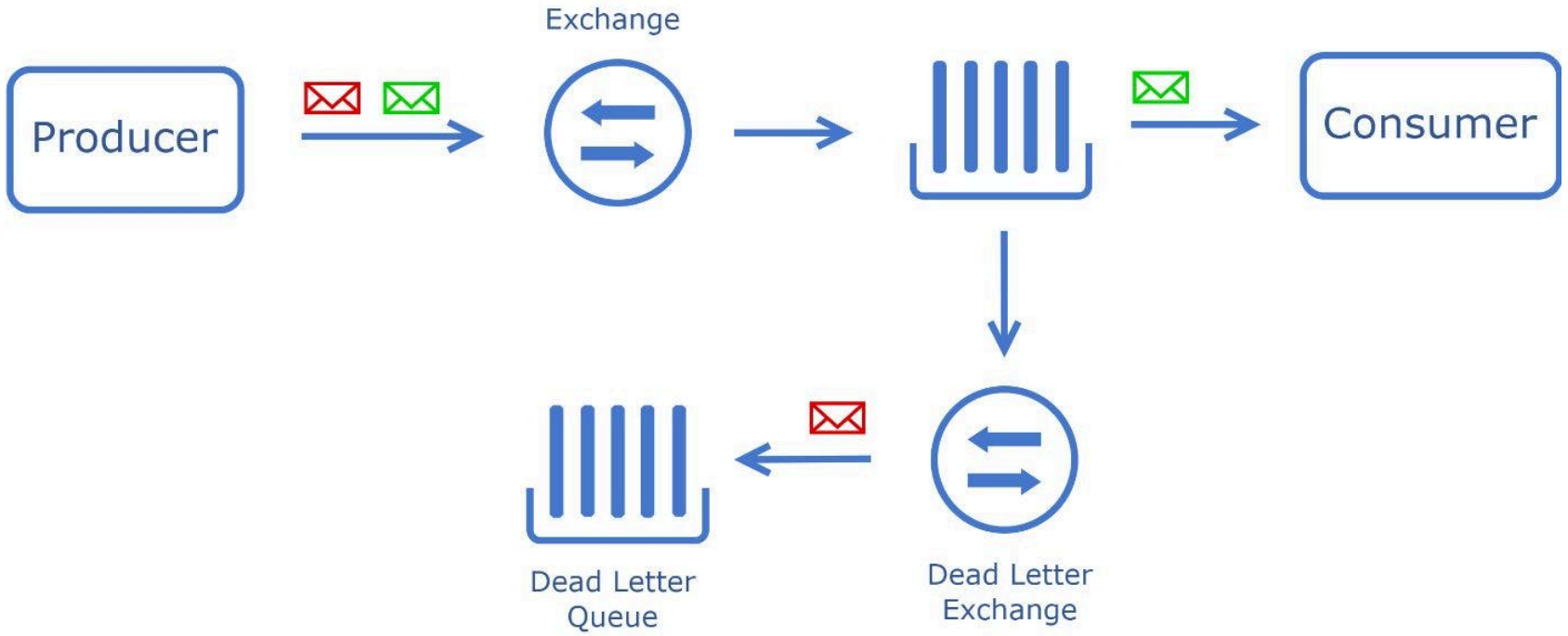
- Doručuje zprávy do všech front, které jsou k němu vázány.

Header exchange



- Doručuje zprávy na základě více atributů vyjádřených jako hlavičky.

Dead Letter



- Fronty mrtvých dopisů shromažďují zprávy, které nemohly být úspěšně zpracovány z různých důvodů, jako jsou chyby při zpracování, vypršení platnosti zprávy nebo problémy s doručením.

Protokoly pro zasílání zpráv

1. AMQP (Advanced Message Queuing Protocol)
2. COAP (Constrained Application Protocol)
3. MQTT (Message Queue Telemetry Transport)
4. STOMP (Simple Text Oriented Messaging Protocol)
5. Kafka protocol
6. ZMTP (ZeroMQ Message Transport Protocol)

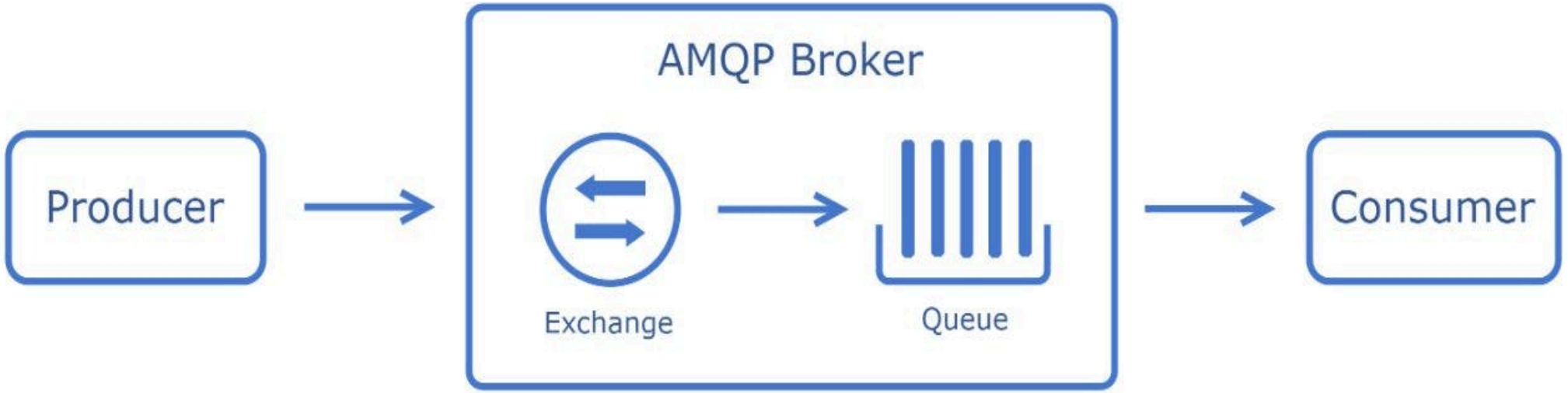
AMQP (Advanced Message Queuing Protocol)

- Otevřený standardní protokol pro aplikace v oblasti finančních systémů, enterprise a business procesů.
- Založen na protokolu TCP s dalšími mechanismy spolehlivosti (at-most-once, at-least-once nebo once-delivery).
- Bezpečnost: TLS/SSL, SASL, PLAIN
- Podporuje jak komunikaci typu point-to-point, tak komunikační paradigmata publish-subscribe.
- Programovatelný protokol: některé entity a schémata směrování jsou primárně definovány aplikacemi.

---

https://www.amqp.org/about/what

AMQP architektura



- Vytváří fronty zpráv, kde zpráva čeká, dokud ji odběratel nepřečte

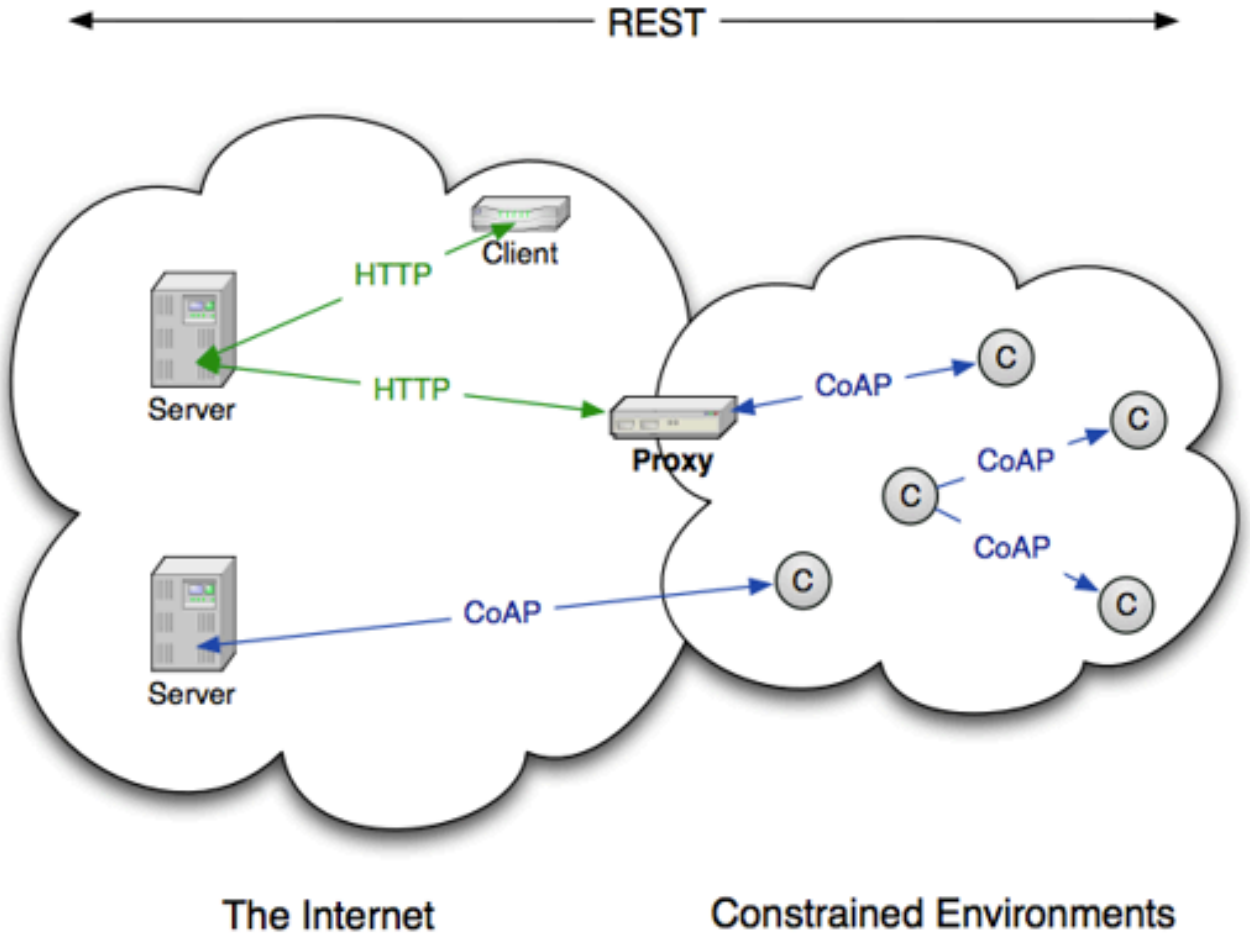
COAP (Constrained Application Protocol)

- Protokol založený na modelu REST
  - Manipuluje se zdroji pomocí stejných metod jako HTTP
- Využívá UDP transportní protokol
  - Režie protokolu TCP je příliš vysoká a jeho řízení toku není vhodné pro krátkodobé transakce.
- UDP má nižší režii a podporuje vícesměrové vysílání, ale datagramy
  - se mohou ztratit
  - mohou být duplikovány
  - mohou dorazit v nesprávném pořadí

---

https://github.com/Tanganelli/CoAPthon

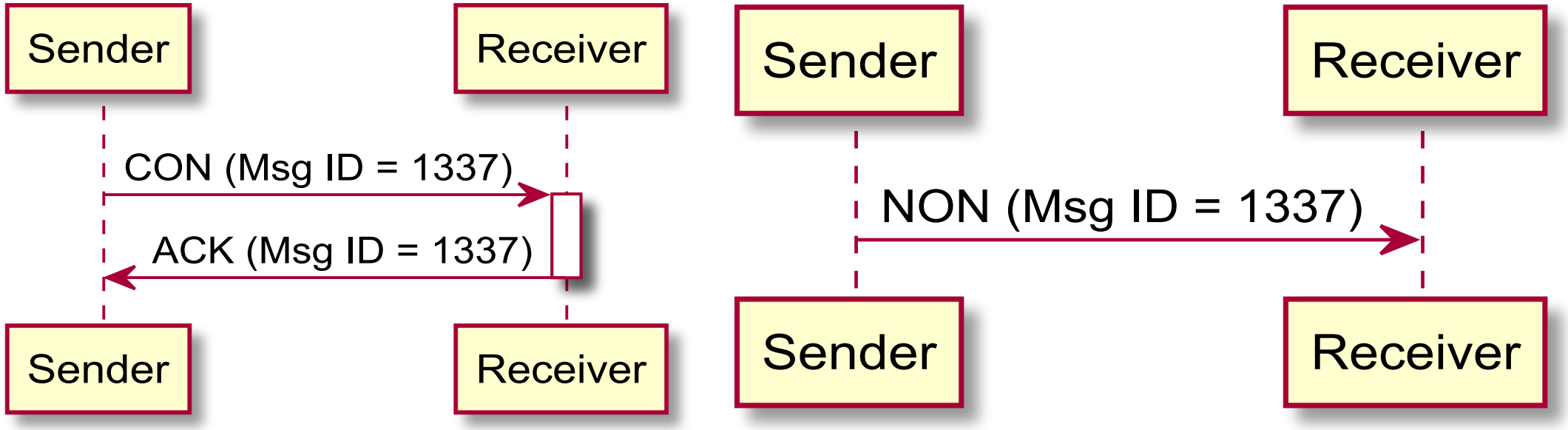
COAP architektura



COAP protokol

- Rozdělen do dvou dílčích vrstev
  - request/response
    - GET, PUT, POST a DELETE
  - zprávy
    - SUCCESS, CLIENT ERROR, SERVER ERROR
- Čtyři typy zpráv:
  - S potvrzením (confirmable) - vyžaduje ACK
  - Bez potvrzení (non-confirmable) - není třeba ACK
  - Potvrzení (ACK)
    - Reset - indikuje, že byla přijata zpráva, ale chybí kontext pro zpracování
    - Prázdná - pouze hlavička o velikosti 4B

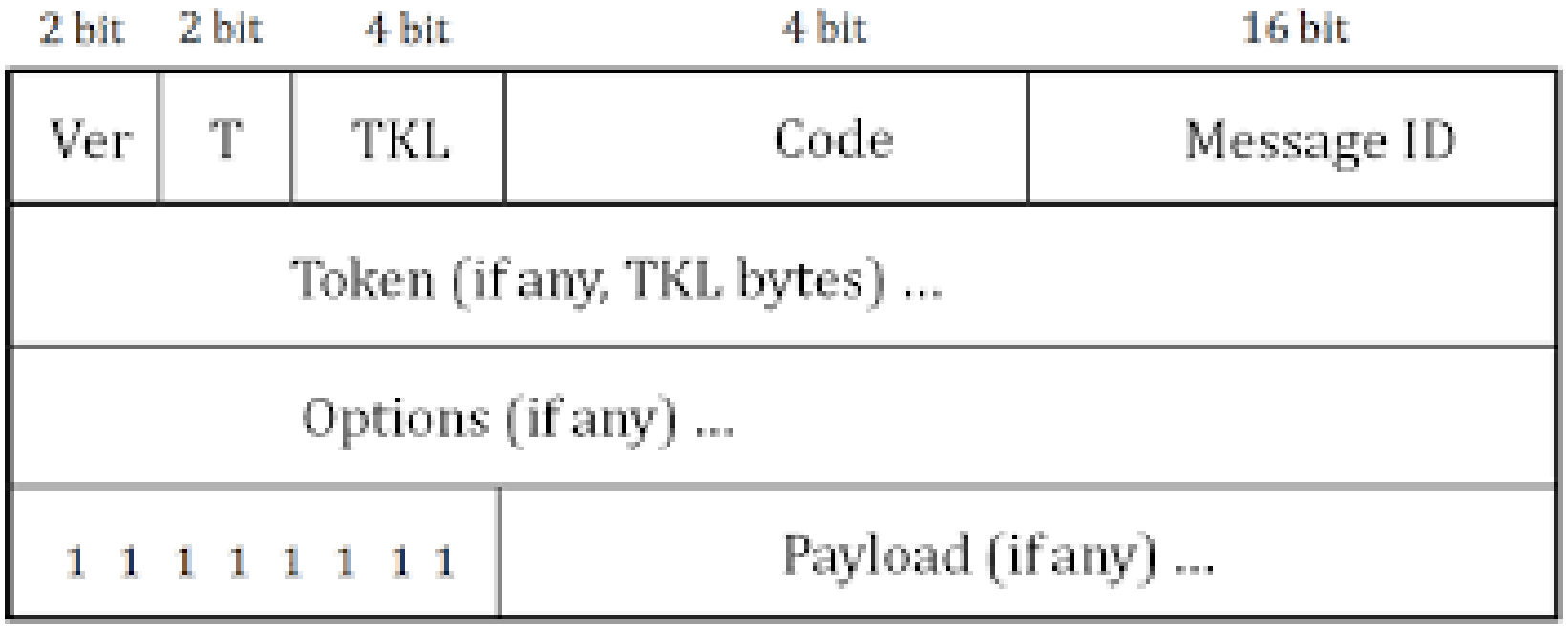
Příklad COAP komunikace



COAP protokol

- Každý dotaz má unikátní URI
- Specifikace a detaily: https://coap.me/
- Rozdíly proti HTTP:
  - Založen na protokolu UDP (ale lze použít volitelné mechanismy pro zvýšení spolehlivosti, tj. potvrzování zpráv + zpětné přenosy).
  - Asynchronní paradigma požadavek/odpověď
  - Jiná (kratší) hlavička paketu (viz další slide)
  - Mechanismy zjišťování služeb (service discovery) a proxy

COAP zpráva



- Token - hodnota pro zjištění korelace mezi dotazem a odpovědí
- Message ID - páruje CON a ACK zprávy

Mechanismy spolehlivosti

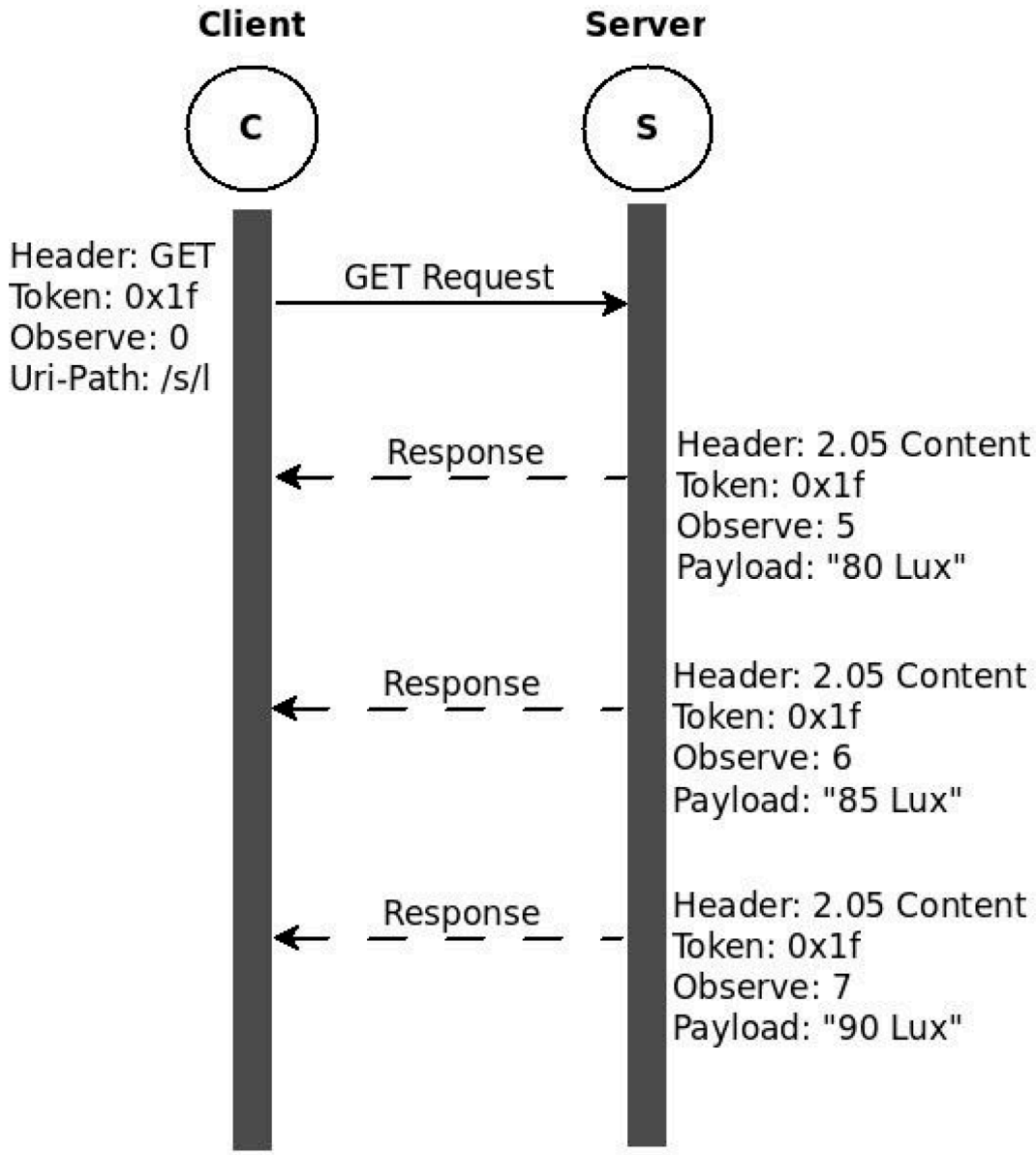
CoAP implementuje některé jednoduché mechanismy spolehlivosti

1. Detekce duplicit pro potvrzené (CON) i nepotvrzené (NON) zprávy
2. Jednoduchá spolehlivost opakovaného přenosu typu stop-and-wait s exponenciálním zpětným přenosem pro potvrzitelné zprávy
    - Odesílatel přeposílá zprávu Confirmable v exponenciálně rostoucích intervalech, dokud neobdrží ACK (nebo zprávu RST) nebo dokud nevyčerpá všechny pokusy.

Odběr dat

Mechanismus OBSERVE umožňuje implementovat mechanismus odběru dat

1. Klient požádá o prostředek (GET) s polem Observe Option.
2. Server přidá klienta do seznamu pozorovatelů daného prostředku.
3. Při každé změně cílového prostředku server informuje všechny jeho pozorovatele.



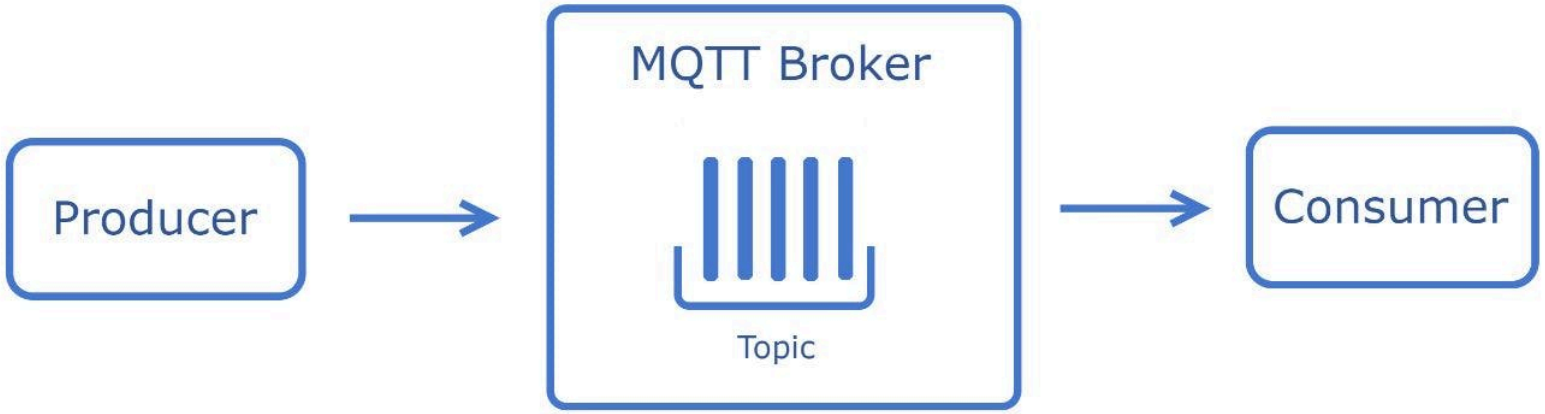
Zabezpečení

- CoAP se při zabezpečení komunikace mezi klientem a serverem spoléhá na protokoly nižší vrstvy.
- Šifrování zpráv na vrstvě TSP (DTLS - Datagram Transport Layer Security) nebo na síťové vrstvě (IPSec).
- Vzhledem k tomu, že protokol CoAP realizuje podmnožinu funkcí protokolu HTTP/1.1, vztahují se bezpečnostní hlediska protokolu HTTP i na protokol CoAP. Kromě toho CoAP představuje některé jedinečné zranitelnosti
    - Proxy servery jsou ze své podstaty man-in-the-middle.
    - Riziko zesílení zpráv a útoků DDoS.
    - Podvržení IP kvůli absenci handshake v UDP.

MQTT (Message Queue Telemetry Transport)

- Protokol pro nenáročné IoT apliace, domácí automatizaci a mobilní komunikace
- Navržen pro zařízení s omezenou šířkou pásma, nespolehlivou konektivitou a sítě s vysokou latencí
- Původně jej navrhli Andy Stanford-Clark (IBM) a Arlen Nipper v roce 1999 pro propojení telemetrických systémů ropovodů přes satelit.
- Je určen pro sítě TCP/IP
  - pro sítě bez TCP existuje implementace MQTT-SN (NS - sensor network)
- Používá datově agnostický textový protokol
- Zajišťuje spolehlivost - poskytuje některé mechanismy pro zajištění doručení

MQTT broker



- Broker zajišťuje QoS a může uchovávat zprávy (data)
  - Vydavatel rozhoduje o tom, zda si broker zprávu ponechá.
  - V tomto případě každý odběratel při odběru automaticky obdrží nejnovější hodnotu, takže Broker udržuje jakýsi "stav".

MQTT data

- Zprávy (obsah) jsou uspořádány do témat ve formě stromové struktury
  - jako je adresářová cesta, oddělovačem je / (lomítko)
- Odběratel se může přihlásit k odběru konkrétního tématu nebo může použít zástupný vzor pro odběr různých témat:
  - [#] znamená celou větev
  - [+] znamená jednoúrovňové

Příklad:

- Vydavatel vydává např: **CVUT/FEL/209/Sensor/Temperature**
- Odběratel se přihlásí k odběru: **CVUT/FEL/+/Senzor/#**
- Odběratel bude upozorněn vždy, když zařízení odešle informaci o jakémkoli měření (tj. teploty, ale také vlhkosti) provedeném **/Senzor/** někde v budově **CVUT/FEL** (může to být místnost, ale také třeba chodba)

MQTT QoS

Level 0 - Neuznaná služba

- Doručeno každému odběrateli maximálně jednou
- Stejné garance doručení jako TCP

Level 1 - Uznávaná služba

- Zajišťuje doručení zprávy alespoň jednou.
- Broker očekává potvrzení, jinak zprávu znovu odešle

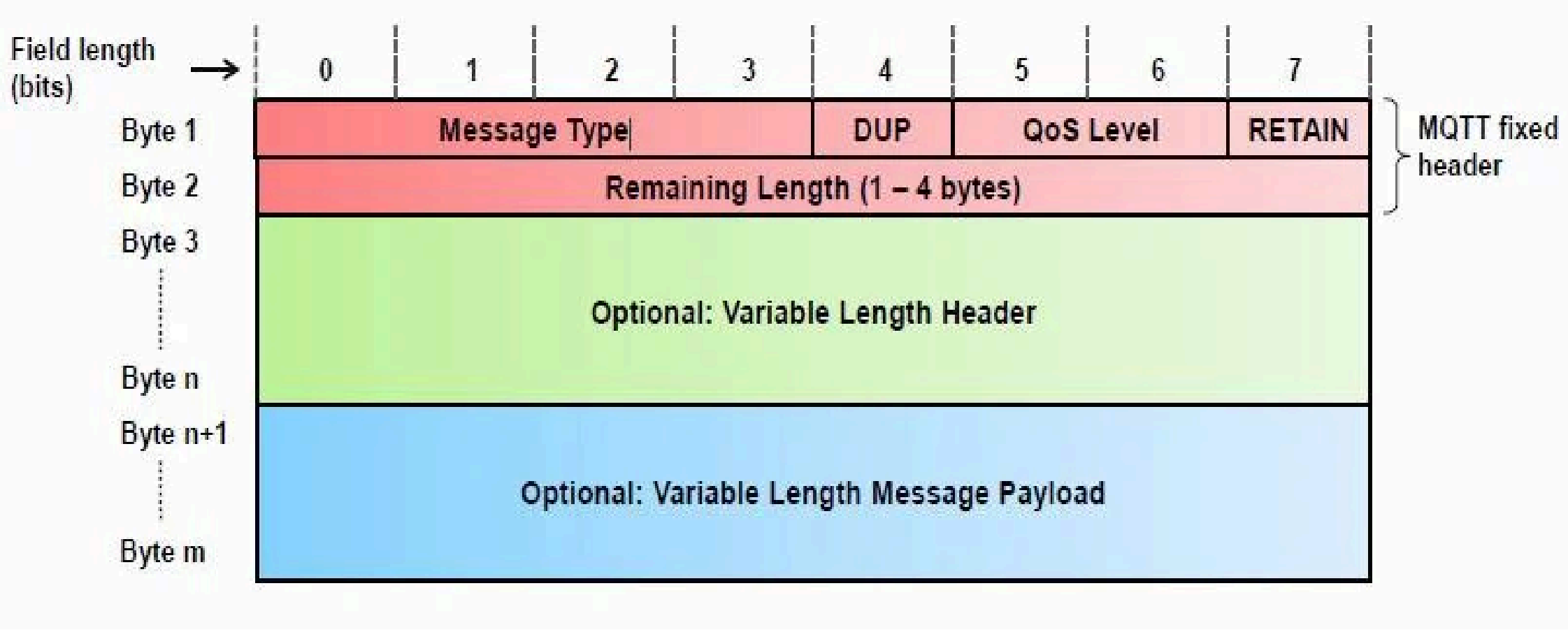
Level 2 - Zajištěná služba

- Dvoustupňové doručení
- Zajišťuje, že zpráva je každému účastníkovi doručena přesně jednou

Další vlastnosti MQTT

- Příznak čisté relace (volitelný) - trvanlivá připojení:
  - Pokud je **true**, Broker odstraní všechny klientské odběry při odpojení klienta.
  - Pokud je false, spojení zůstane nečinné a všechny zprávy se shromažďují (QoS v závislosti na typu připojení) a doručeny, jakmile je připojení obnoveno.
- Klient může brokerovi nařídit, aby ho nechal odeslat konkrétní téma (nebo témata), když se objeví neočekávané spojení.
  - Zjištění selhání/havárie: vhodné např. pro kritické a bezpečnostní systémy
- Bezpečnost
  - Slabá - uživatelská jména a hesla zasílána v prostém textu
  - Lze využít zabezpečený kanál (SSL/TLS)

MQTT protokol



Typy MQTT zpráv

MQTT message	Description with direction of flow
CONNECT	C->S, the client requests to connect to server
CONNACK	S->C, connect acknowledgment
PUBLISH	C->S OR S->C Publish message
PUBACK	C->S OR S->C Publish acknowledgment
PUBREC	C->S OR S->C Publish received
PUBREL	C->S OR S->C Publish release
PUBCOMP	C->S OR S->C Publish complete

MQTT message	Description with direction of flow
SUBSCRIBE	C->S Client subscribe request
SUBACK	S->C Subscribe acknowledgment
UNSUBSCRIBE	C->S Unsubscribe request
UNSUBACK	S->C Unsubscribe acknowledgment
PINGREQ	C->S Ping request
PINGRESP	S->C PING RESPONSE
DISCONNECT	C->S Client is disconnecting

MQTT subscribe v Pythonu

```
import paho.mqtt.client as paho

def on_message(mosq, obj, msg):
    print ("%-20s %d %s" % (msg.topic, msg.qos, msg.payload))
    # mosq.publish('pong', 'ack', 0)

def on_publish(mosq, obj, mid):
    pass

if __name__ == '__main__':
    client = paho.Client()
    client.on_message = on_message
    # client.on_publish = on_publish

    #client.tls_set('root.ca', certfile='c1.crt', keyfile='c1.key')
    client.connect("127.0.0.1", 1883, 60)

    # client.subscribe("kids/yolo", 0)
    # client.subscribe("adult/#", 0)
    client.subscribe("#", 0)
```

MQTT publish v Pythonu

```
import paho.mqtt.publish as publish

msgs = [{'topic': "CVUT/FEL/209/Temperature", 'payload': "23"},
        {'topic': "CVUT/FEL/209/Humidity", 'payload': "60"}]

host = "localhost"

if __name__ == '__main__':
    # publish a single message
    publish.single(topic="CVUT/FEL/209/test", payload="test", hostname=host)

    # publish multiple messages
    publish.multiple(msgs, hostname=host)
```

MQTT ve Flasku

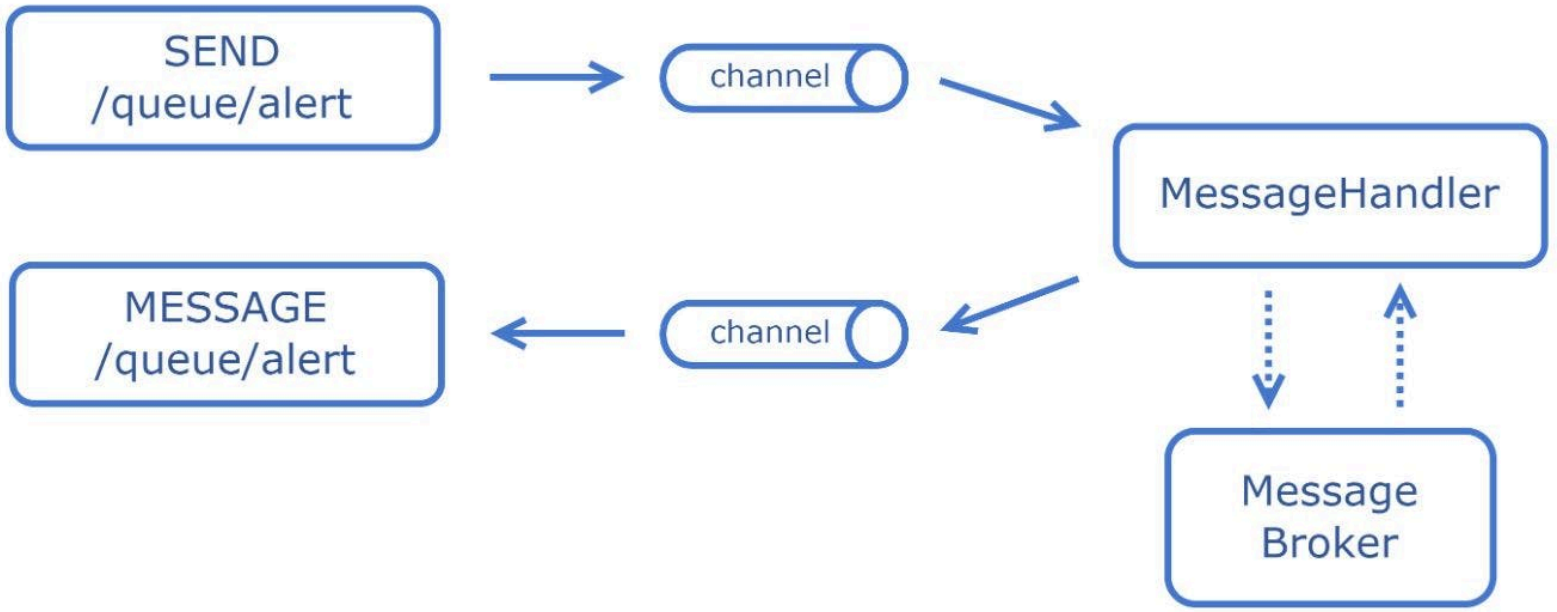
```
from flask import Flask
from flask_mqtt import Mqtt

app = Flask(__name__)
# use the free broker from HIVEMQ
app.config['MQTT_BROKER_URL'] = 'broker.hivemq.com'
# default port for non-tls connection
app.config['MQTT_BROKER_PORT'] = 1883
# set the username here if you need authentication for the broker
app.config['MQTT_USERNAME'] = ''
# set the password here if the broker demands authentication
app.config['MQTT_PASSWORD'] = ''
# set the time interval for sending a ping to the broker to 5 seconds
app.config['MQTT_KEEPALIVE'] = 5
# set TLS to disabled for testing purposes
app.config['MQTT_TLS_ENABLED'] = False

mqtt = Mqtt()
```

MQTT subscribe/publish ve Flasku

```
mqtt.subscribe('home/mytopic')

@mqtt.on_connect()
def handle_connect(client, userdata, flags, rc):
    mqtt.subscribe('home/mytopic')

@mqtt.on_message()
def handle_mqtt_message(client, userdata, message):
    data = dict(
        topic=message.topic,
        payload=message.payload.decode()
    )

mqtt.unsubscribe('home/mytopic')
mqtt.unsubscribe_all()
```

```
mqtt.publish('home/mytopic', 'hello world')
```

STOMP (Simple Text Oriented Messaging Protocol)



- Jednoduchý, na textu založený protokol, který se snadno implementuje a je vhodný pro scénáře, kde pokročilé funkce zasílání zpráv nejsou prioritou.

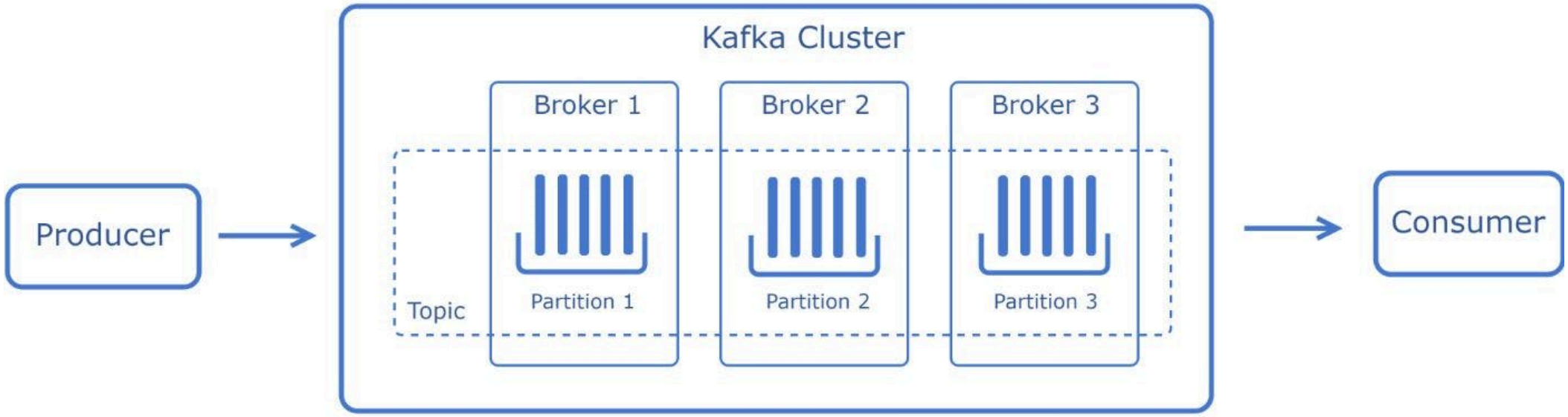
STOMP (Simple Text Oriented Messaging Protocol)

- **Případy použití:** Prostředí pro rychlý vývoj a jednoduché aplikace pro zasílání zpráv
- **Model zasílání zpráv:** point-to-point, publish-subscribe
- **Zabezpečení:** PLAIN; spoléhá se na základní transportní protokol.
- **Adresování:** Adresace založená na rámcích se záhlavími pro cíl, typ obsahu atd.
- **Architektura:** broker

---

https://stomp.github.io/

Kafka Protocol

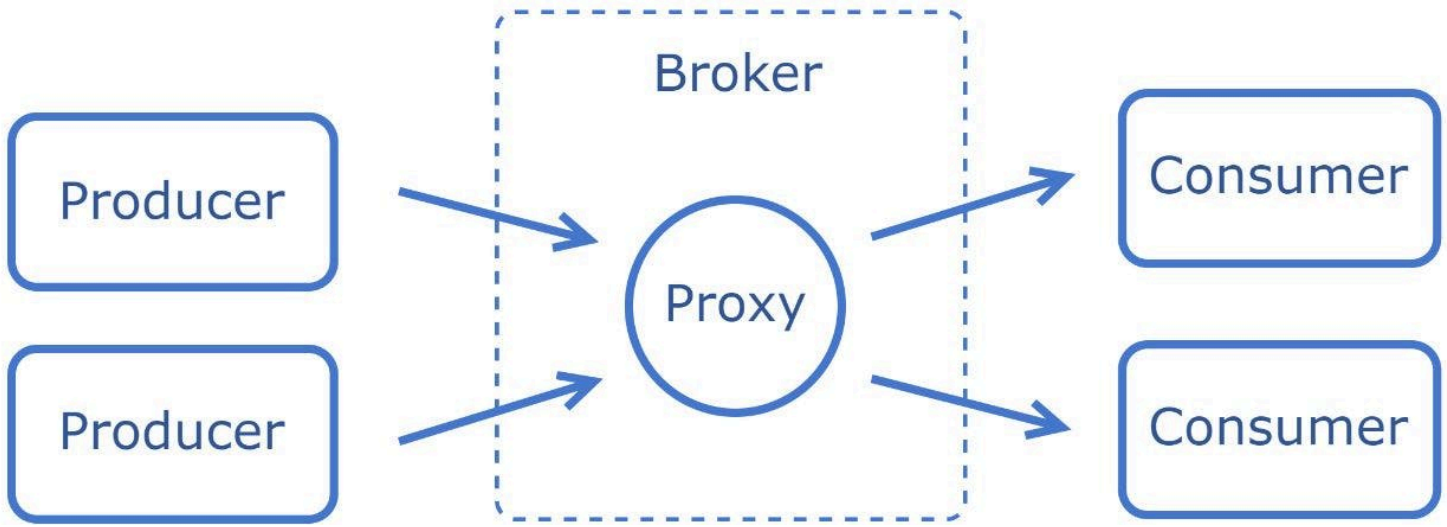


- Spojen s Apache Kafka, distribuovanou streamovací platformou schopnou zpracovávat datové toky s vysokou propustností.

Kafka Protocol

- **Případy použití:** Analýza v reálném čase, datové pipelines, aplikace pro zpracování datových toků
- **Model zasílání zpráv:** publish-subscribe
- **Zabezpečení:** SSL/TLS, SASL
- **Adresování:** Adresace založená na tématech s rozdělením pro škálovatelnost.
- **Architektura:** Distribuovaná architektura systému se zprostředkovateli a koordinací.

ZMTP (ZeroMQ Message Transport Protocol)



- Vysoce výkonná knihovna pro asynchronní zasílání zpráv pro vytváření škálovatelných distribuovaných aplikací.

ZMTP (ZeroMQ Message Transport Protocol)

- **Případy použití:** Aplikace s vysokou propustností a nízkou latencí, architektura mikroslužeb.
- **Model přenosu zpráv:** request-response, publish-subscribe, pipeline, exclusive pair atd.
- **Zabezpečení:** PLAIN, CurveZMQ a ZAP.
- **Adresování:** Flexibilní adresování pomocí soketů
- **Architektura:** Založená na knihovně, umožňující návrh bez zprostředkovatele nebo různé zprostředkované konfigurace.

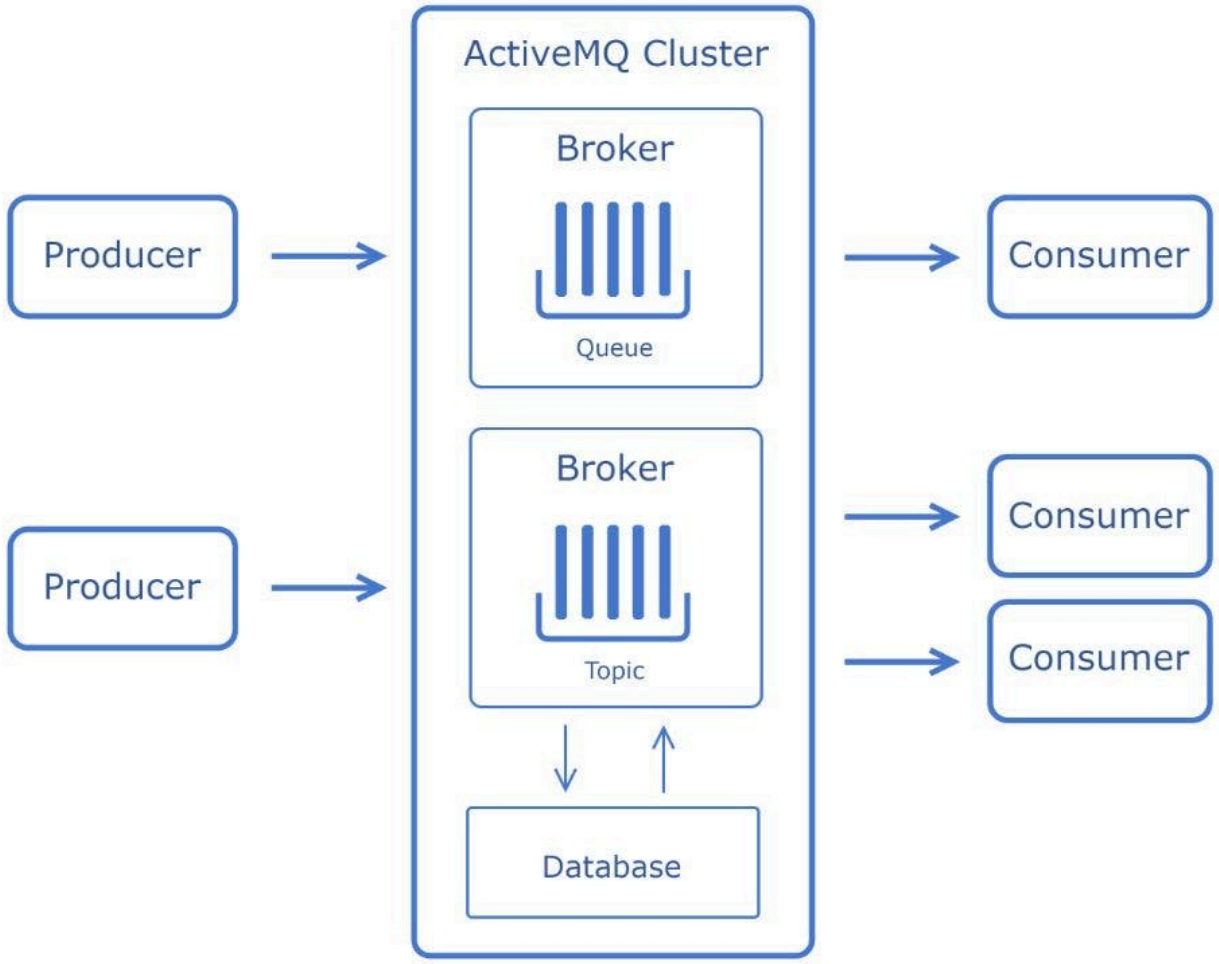
Brokery

1. ActiveMQ
2. RabbitMQ
3. Apache Kafka
4. ZeroMQ
5. Další brokery

ActiveMQ

- Open-source, víceprotokolový zprostředkovatel zpráv založený na jazyce Java navržený společností Apache.
- Robustnosní a flexibilní, podporuje různé protokoly pro zasílání zpráv a klienty.

https://activemq.apache.org/



- Podporuje širokou škálu protokolů pro zasílání zpráv, včetně AMQP, MQTT, OpenWire, STOMP a JMS (Java Message Service).
- Využívá architekturu brokerů, kde centrální broker zajišťuje směrování, doručování a řazení zpráv do fronty.
- Nabízí možnosti perzistence zpráv, včetně databázového úložiště (pro trvanlivost) a úložiště souborového systému, které podporuje scénáře s vysokým výkonem i trvanlivostí.
- Podporuje clustering a vyvažování zátěže, což umožňuje vysokou dostupnost a škálovatelnost.
- Poskytuje různé možnosti potvrzování zpráv, čímž zvyšuje spolehlivost zpráv.

Nevýhody

- ActiveMQ je sice robustní, ale nemusí se vyrovnat výkonu některých novějších zprostředkovatelů zpráv, zejména ve scénářích s extrémně vysokými požadavky na propustnost.
- Může být obtížné konfigurovat a spravovat, zejména v clusterových konfiguracích.
- Může vyžadovat značné prostředky, zejména při velkém zatížení, pro optimální výkon.
- V porovnání s některými novějšími zprostředkovateli mohou být nástroje pro správu méně komplexní a uživatelsky přívětivé.

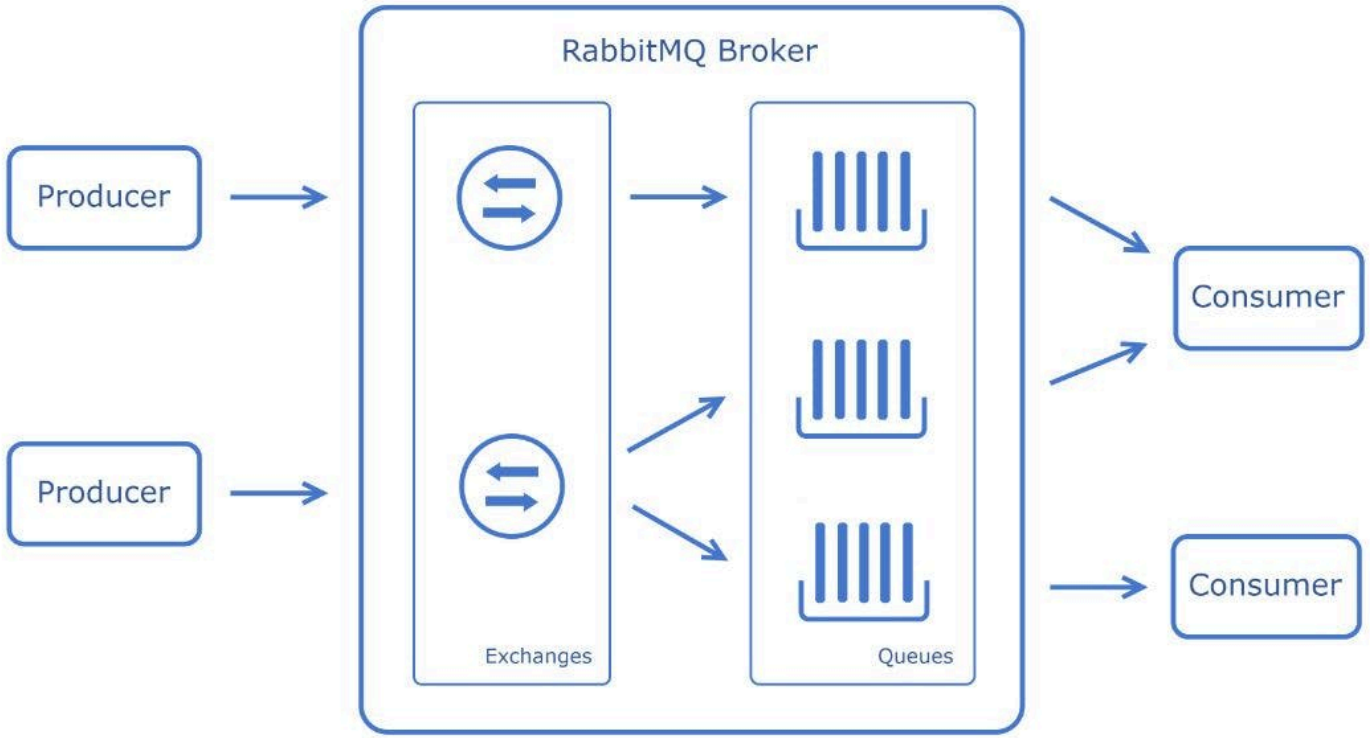
AMPQ v Pythonu

```
import time
import sys
import stomp

# Create a Listener class inheriting the stomp.ConnectionListener class
class Listener(stomp.ConnectionListener):
# Override the methods on_error and on_message provides by the parent class
    def on_error(self, headers, message):
        print('received an error "%s"' % message)

    def on_message(self, headers, message):
        print('received a message "%s"' % message)

# Decare hosts as an array of tuples containing the ActiveMQ server IP
# address or hostname and the port number
hosts = [('localhost', 61616)]

# Create a connection object by passing the hosts as an argument
conn = stomp.Connection(host_and_ports=hosts)
```

```
# Tell the connection object to listen for messages using the Listener class we created above
conn.set_listener('', Listener())

# Initiate the connection with the credentials of the ActiveMQ server
conn.start()
conn.connect('admin', 'admin', wait=True)

# Register a consumer with ActiveMQ. This tells ActiveMQ to send all
# messages received on the queue 'queue-1' to this listener
conn.subscribe(destination='/queue/queue-1', id=1, ack='auto')

# Act as a message producer and send a message the queue queue-1
# The actual message to be sent is picked up from the command line arguments
conn.send(body=' '.join(sys.argv[1:]), destination='/queue/queue-1')

# When this message is received by the listener, it will be handled
# by the on_message method we defined above.
time.sleep(2)
conn.disconnect()
```

RabbitMQ

- Je napsán v jazyce Erlang.
- Postaven na frameworku Open Telecom Platform.
- Asynchronní komunikace mezi distribuovanými systémy prostřednictvím různých protokolů, především pak AMQP.



https://www.rabbitmq.com/

- Kromě protokolu AMQP podporuje prostřednictvím zásuvných modulů také protokoly MQTT, STOMP a další.
- Broker v modelu Publish-Subscribe
- Vazba Exchange-Queue: Zprávy v RabbitMQ jsou publikovány do výměn, které jsou pak směrovány do vázaných front na základě směrovacích klíčů a vzorů.
- Podporuje trvalé (trvalé na disku) a přechodné (v paměti) zprávy.
- Pro vysokou dostupnost a škálovatelnost lze RabbitMQ clusterovat a rozdělit fronty mezi více uzlů.
- Nabízí několik typů výměny (například přímou, tematickou, fanout a hlavičky) pro různorodou logiku směrování.
- Podporuje zásuvné moduly ověřování, včetně LDAP.

Nevýhody

- Křivka učení: Pochopení směrování a nastavení RabbitMQ může být pro začátečníky složité.
- Využití paměti: Může být náročný na paměť, zejména při velkém zatížení, což vyžaduje řádné monitorování a ladění.
- Závislost na Erlangu: Jelikož je postaven na Erlangu, zavádí další technologický stack, se kterým se týmy budou muset seznámit.
- Výkon při vysokém zatížení: Přestože je obecně výkonný, může být nutné vyladit výkon při extrémně vysokém zatížení nebo ve složitých scénářích směrování.

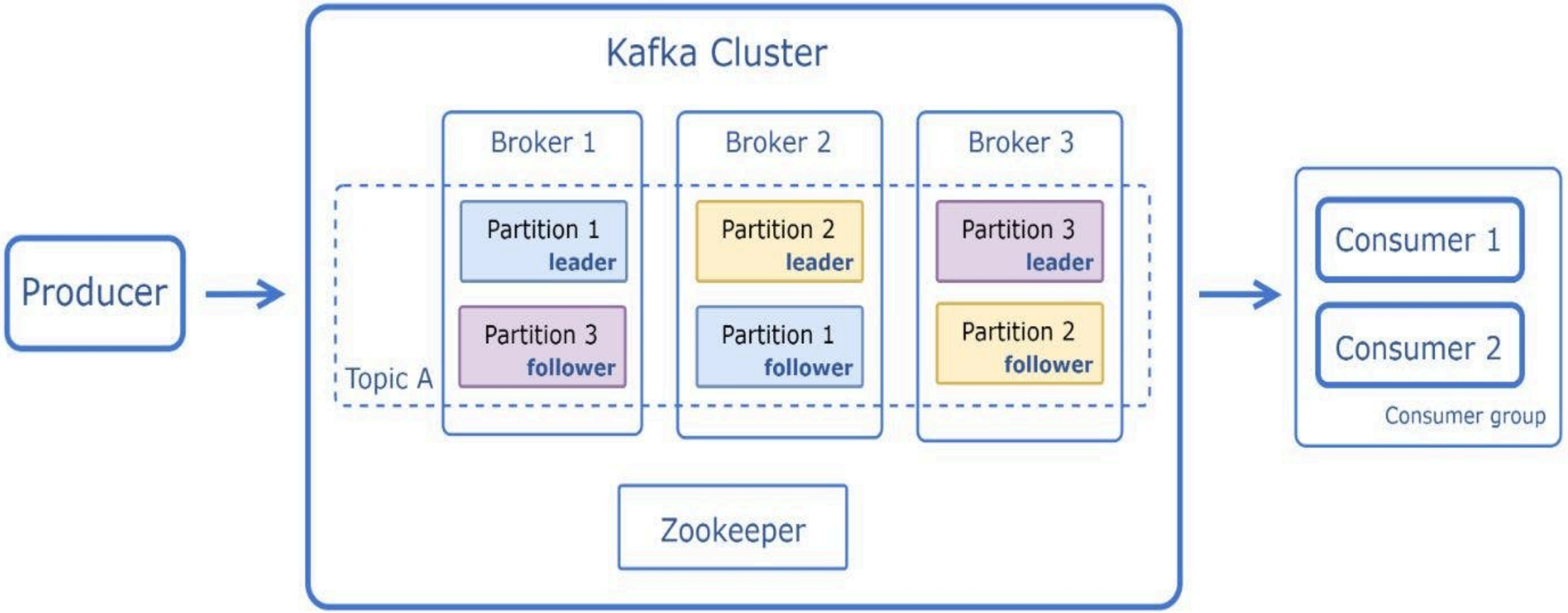
Python Producer

```
import pika

credentials = pika.PlainCredentials('tester','secretPass')

connection = pika.BlockingConnection(pika.ConnectionParameters
  (host='localhost', credentials = credentials))

channel = connection.channel()
channel.exchange_declare('test', durable=True, exchange_type='topic')
channel.queue_declare(queue='A')
channel.queue_bind(exchange='test', queue='A', routing_key='A')
channel.queue_declare(queue='B')
channel.queue_bind(exchange='test', queue='B', routing_key='B')

# messaging to queue named B
message= 'hello consumer!!!!!'
channel.basic_publish(exchange='test', routing_key='B', body= message)
channel.close()
```

Python Consumer

```
import pika

credentials = pika.PlainCredentials('tester','secretPass')
connection = pika.BlockingConnection(pika.ConnectionParameters
(host='localhost', port='5672', credentials= credentials))

channel = connection.channel()
channel.exchange_declare('test', durable=True, exchange_type='topic')

def callbackQueueA(ch,method,properties,body):
  print('Got a message from Queue A: ', body)

def callbackQueueB(ch,method,properties,body):
  print('Got a message from Queue B: ', body)

channel.basic_consume(queue='A', on_message_callback=callbackQueueA, auto_ack=True)
channel.basic_consume(queue='B', on_message_callback=callbackQueueB, auto_ack=True)

#this will be command for starting the consumer session
channel.start_consuming()
```

Apache Kafka



Apache Kafka

- Open-source softwarová platforma pro zpracování datových toků vyvinutá společností LinkedIn a později darovaná nadaci Apache Software Foundation.
- Je navržena pro zpracování velkých objemů dat a umožňuje zpracování dat v reálném čase. Kafka je distribuovaná, rozdělená a replikovaná služba protokolů revizí.

---

https://kafka.apache.org/
https://kafka-python.readthedocs.io/en/master/
https://www.root.cz/clanky/apache-kafka-distribuovana-streamovaci-platforma/

- Broker modelu Publish-Subscribe
- Data v Kafce jsou rozdělena do témat.
  - Každé téma lze rozdělit na oddíly, což umožňuje paralelní zpracování dat.
  - Oddíly také umožňují horizontální škálování Kafky.
- Kafka běží jako cluster na jednom nebo více serverech a cluster Kafka ukládá proudy záznamů do kategorií nazývaných témata.
- Kafka replikuje data ve více uzlech (brokerech), aby byla zajištěna odolnost proti chybám. Pokud dojde k selhání některého uzlu, lze data načíst z ostatních uzlů.
- Kafka používá pro správu a koordinaci clusteru nástroj ZooKeeper, který zajišťuje konzistenci v celém clusteru.
- Kafka ukládá všechna data jako posloupnost záznamů (neboli commit log), čímž zajišťuje trvalé uložení zpráv.

Aplikace

- Zpracování dat v reálném čase: Ideální pro analytické a monitorovací systémy v reálném čase, kde je rychlé zpracování dat klíčové.
- Získávání událostí: Vhodné pro záznam posloupnosti událostí v aplikacích.
- Agregace protokolů: Efektivní pro sběr a zpracování protokolů z více služeb.
- Zpracování datových toků: Lze použít pro komplexní úlohy zpracování datových toků, jako je agregace datových toků nebo filtrování v reálném čase.
- Integrace s technologiemi pro zpracování velkých objemů dat: Často se používá s nástroji pro zpracování a analýzu velkých dat.

Python Consumer

- https://kafka-python.readthedocs.io/en/master/apidoc/KafkaConsumer.html

```
from kafka import KafkaConsumer
consumer = KafkaConsumer('my_favorite_topic')
for msg in consumer:
  print (msg)
```

```
# join a consumer group for dynamic partition assignment and offset commits
from kafka import KafkaConsumer
consumer = KafkaConsumer('my_favorite_topic', group_id='my_favorite_group')
for msg in consumer:
  print (msg)
```

Python Producer

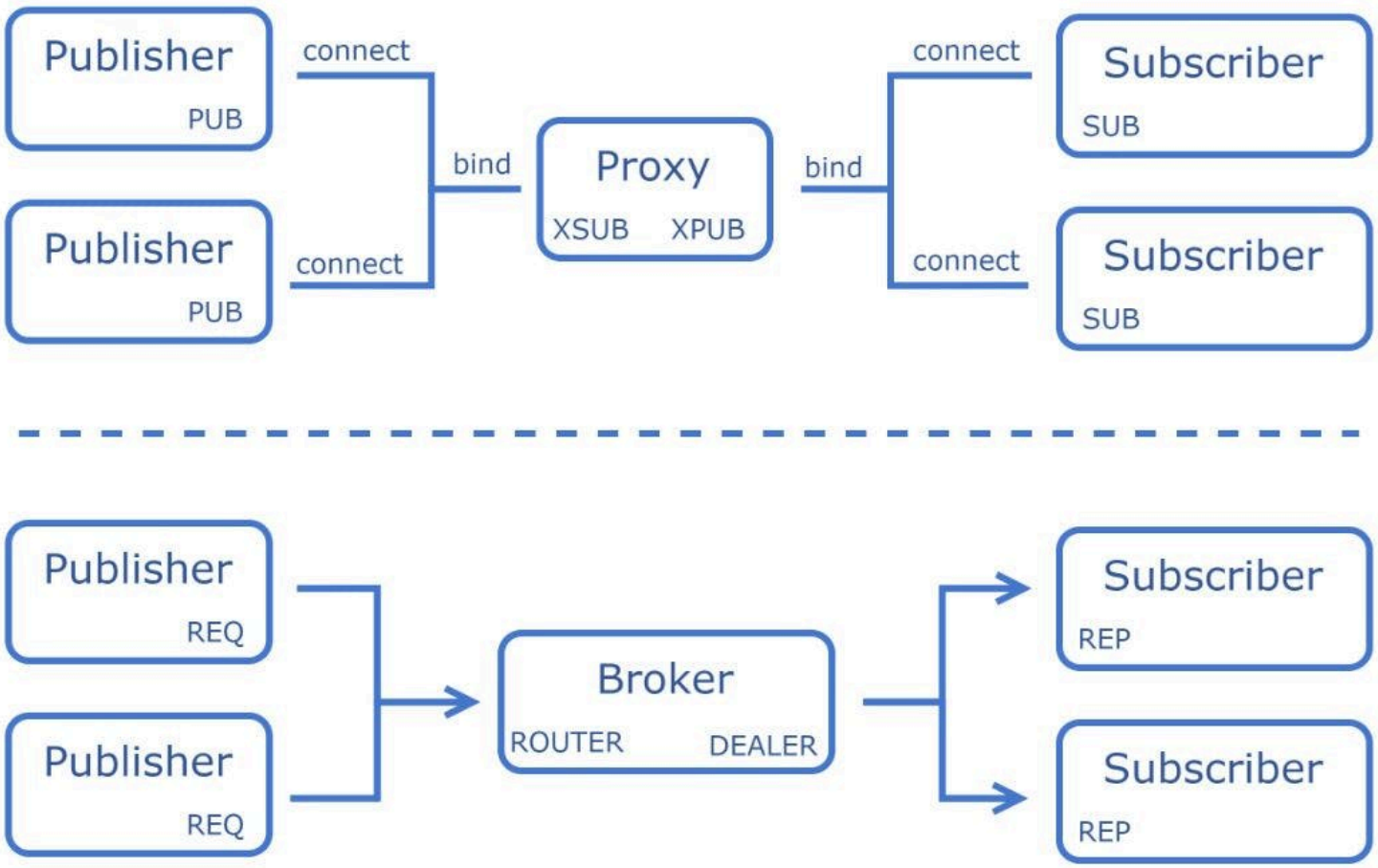
- https://kafka-python.readthedocs.io/en/master/apidoc/KafkaProducer.html

```
from kafka import KafkaProducer
producer = KafkaProducer(bootstrap_servers='localhost:1234')
for _ in range(100):
  producer.send('foobar', b'some_message_bytes')
```

```
# Block until a single message is sent (or timeout)
future = producer.send('foobar', b'another_message')
result = future.get(timeout=60)
```

```
# Use a key for hashed-partitioning
producer.send('foobar', key=b'foo', value=b'bar')
```

ZeroMQ


Publisher
PUB
Publisher
PUB
connect
connect
bind
Proxy
XSUB
XPUB
bind
connect
connect
Subscriber
SUB
Subscriber
SUB
Publisher
REQ
Publisher
REQ
Broker
ROUTER
DEALER
Subscriber
REP
Subscriber
REP

ZeroMQ

- Založený na socketech, nízkoúrovňovém síťovém programování.
- Komunikační modely publish-subscribe, request-reply a fan-out.
- Nepoužívá brokera, což umožňuje přímou komunikaci mezi koncovými body bez nutnosti centrálního zprostředkovatele zpráv.
- Škálovatelné správa vláken s komunikací založenou na soketech.
- Podporuje neblokující asynchronní I/O operace, což je zásadní pro vytváření rychlých a vysoce výkonných aplikací.

---

https://zeromq.org/get-started/
https://zeromq.org/languages/python/

Python server

```
import time
import zmq  # pyzmq

context = zmq.Context()
socket = context.socket(zmq.REP)
socket.bind("tcp://*:5555")

while True:
    #  Wait for next request from client
    message = socket.recv()
    print("Received request: %s" % message)

    #  Do some 'work'
    time.sleep(1)

    #  Send reply back to client
    socket.send(b"World")
```

Python client

```
import zmq

context = zmq.Context()

#  Socket to talk to server
print("Connecting to hello world server…")
socket = context.socket(zmq.REQ)
socket.connect("tcp://localhost:5555")

#  Do 10 requests, waiting each time for a response
for request in range(10):
    print("Sending request %s …" % request)
    socket.send(b"Hello")

    #  Get the reply.
    message = socket.recv()
    print("Received reply %s [ %s ]" % (request, message))
```

Další brokery

- **Redis (Remote Dictionary Server)**
  - distribuované úložiště dat key-value udržující všechny informace v paměti.
  - lze používat jako databázi, cache, nebo message broker.
- **IronMQ**
  - cloudová rešení, výkonný a škálovatný.
- **Microsoft Azure Service Bus**
  - cloudová verze message brokeru pro prostředí Microsoft Azure.
- **Eclipse Mosquitto MQTT Broker**
  - message broker často používaný pro IoT, používá MQTT.