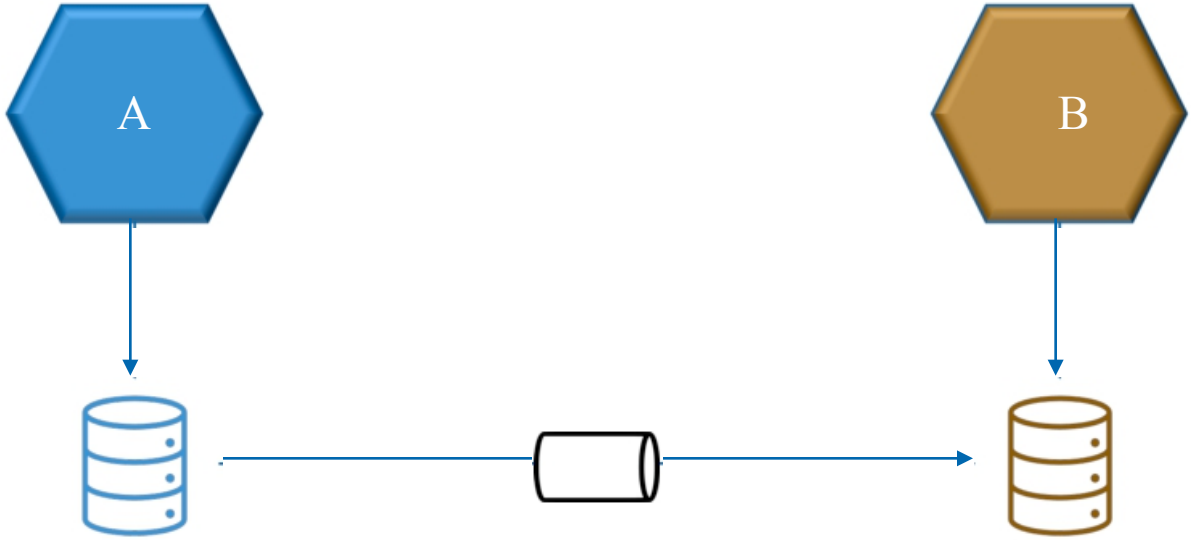
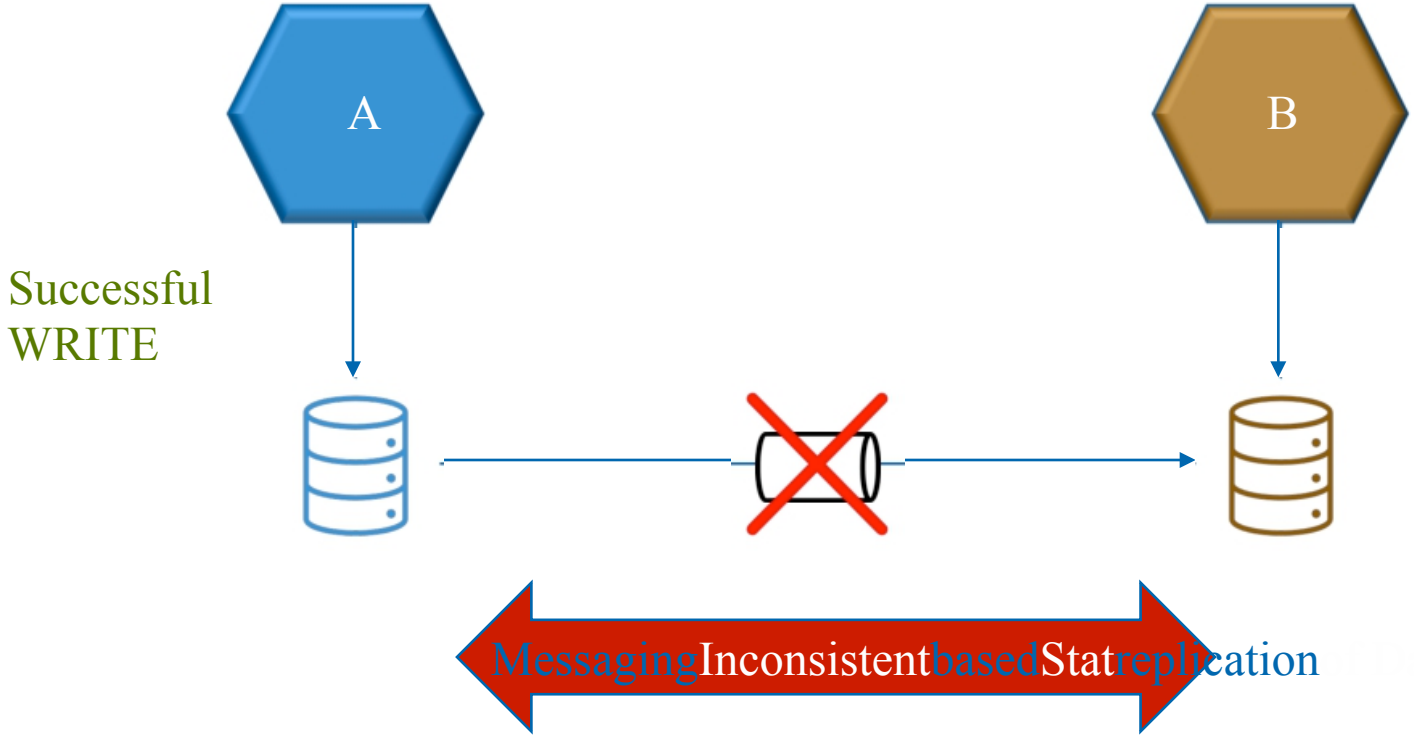
Managing Data Integrity

Data Replication

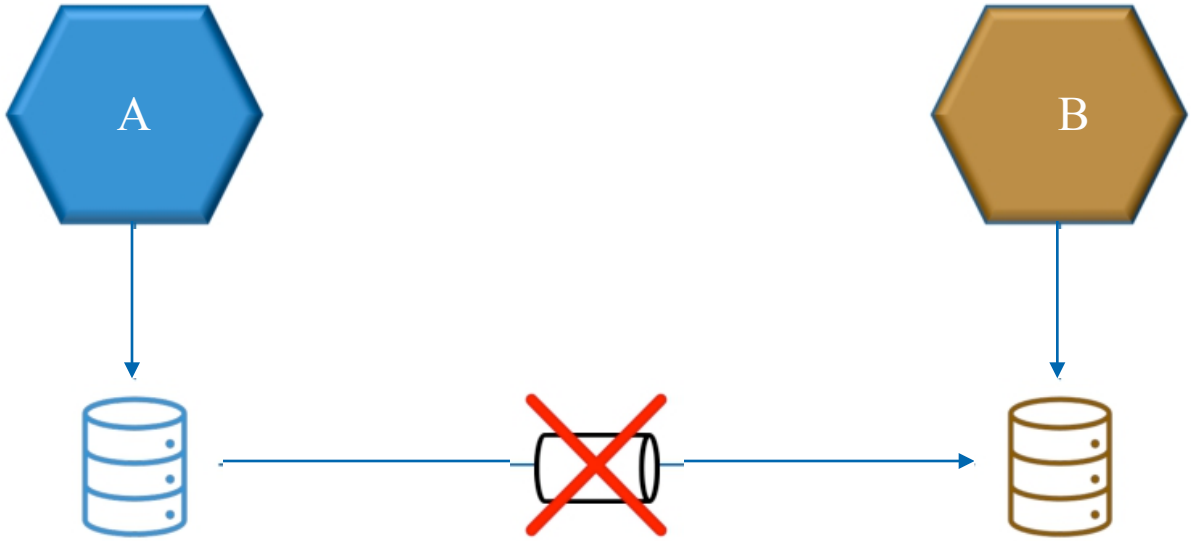


Messaging based replication

Loss of Message



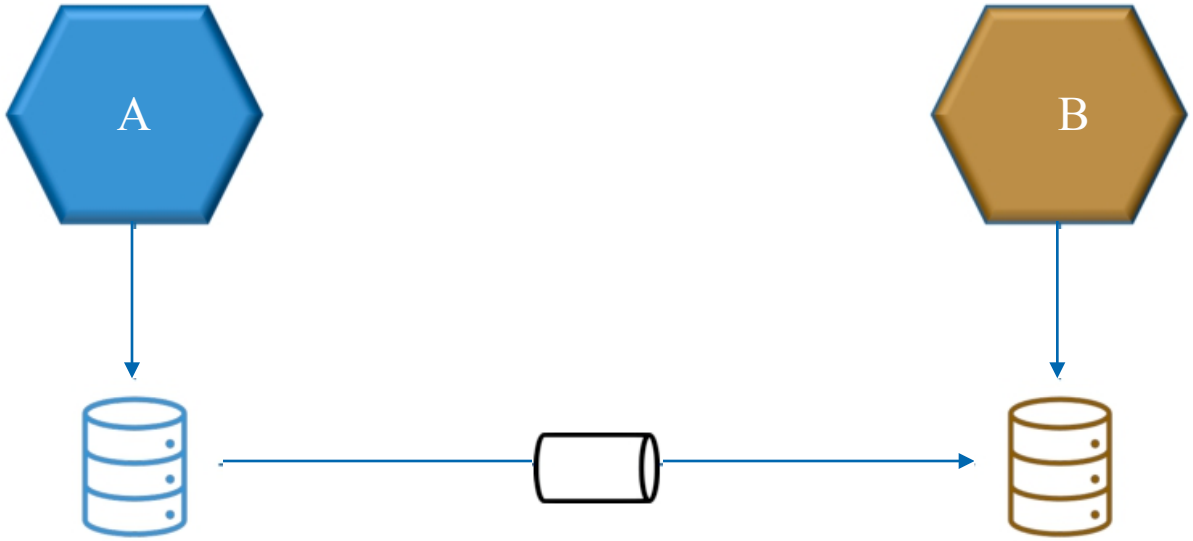
Reliable Messaging pattern



Guaranteed delivery of message to the messaging broker

Managing Data Integrity

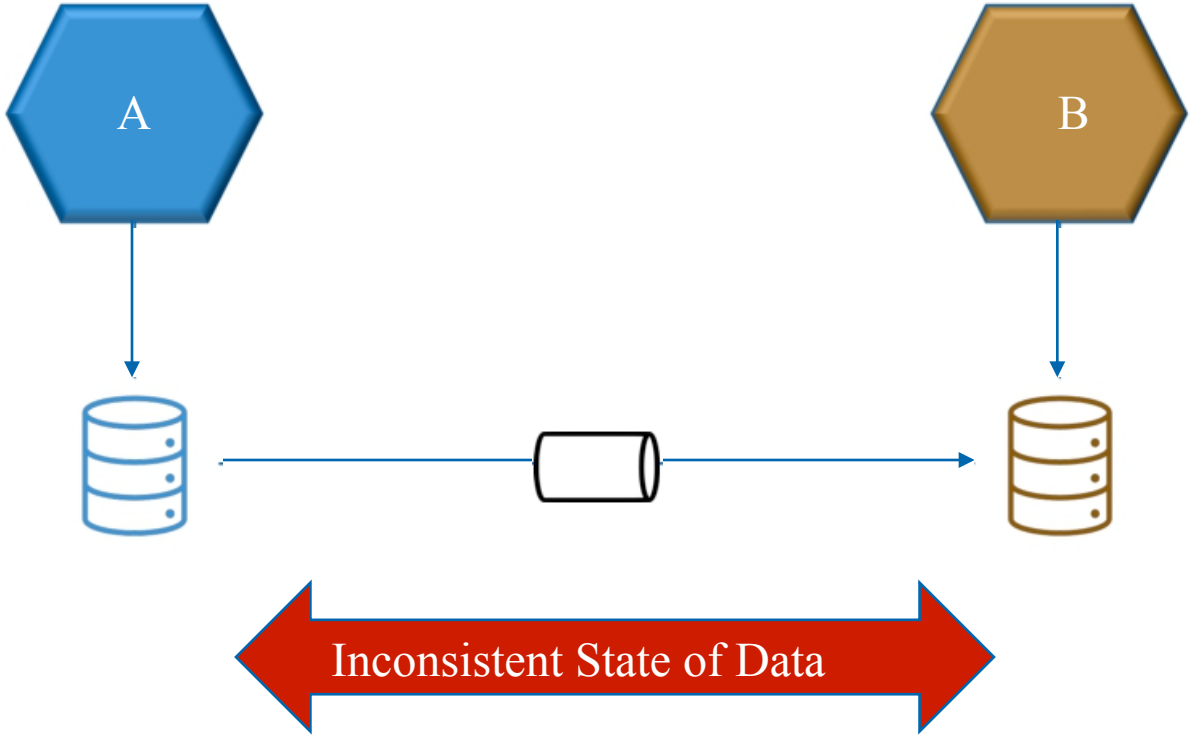
Reliable Messaging pattern



Data will be Eventually Consistent across the database instances

Managing Data Integrity

Producer may send duplicate messages



Managing Data Integrity

Section objective

1. Design for failure
2. Reliable Messaging pattern
3. Duplicate message handling
4. Case Study : Address ACME Proposal CQRS design flaw

Designing for failure

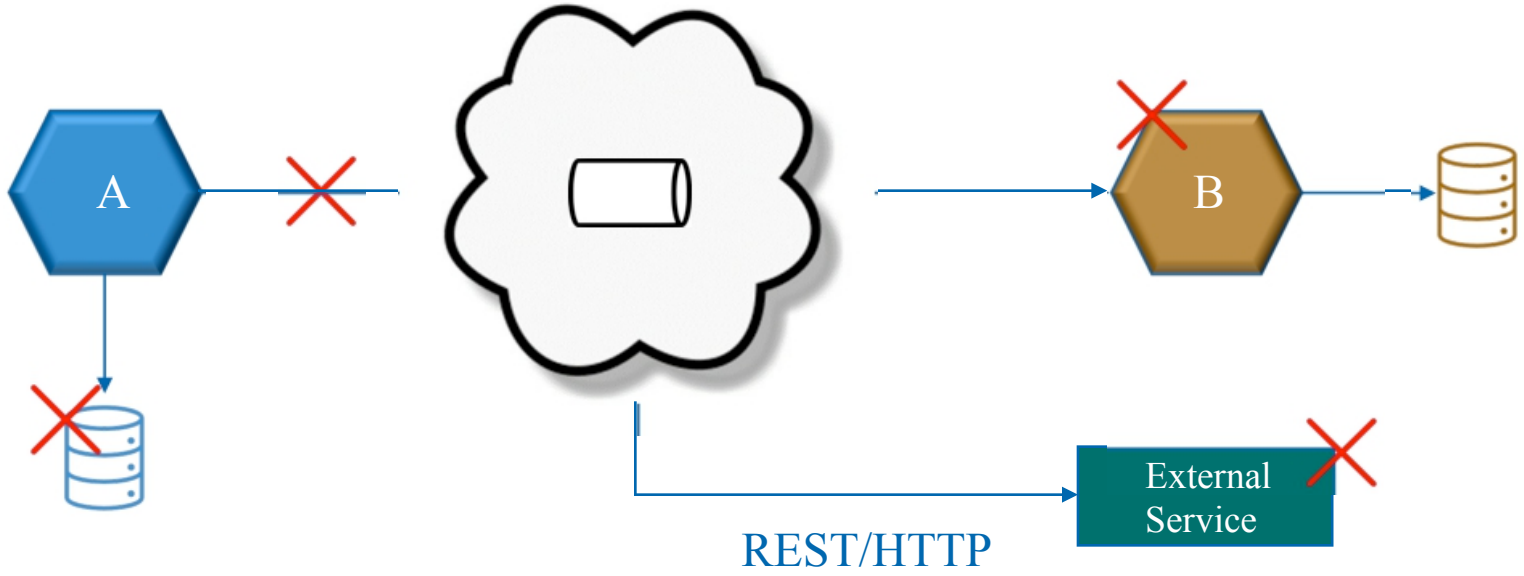
Addressing Message loss due to MQ failure

1. Concept of "Designing for failure"
2. 2 Phase Commit
3. Reliable Messaging Pattern

Design for failure

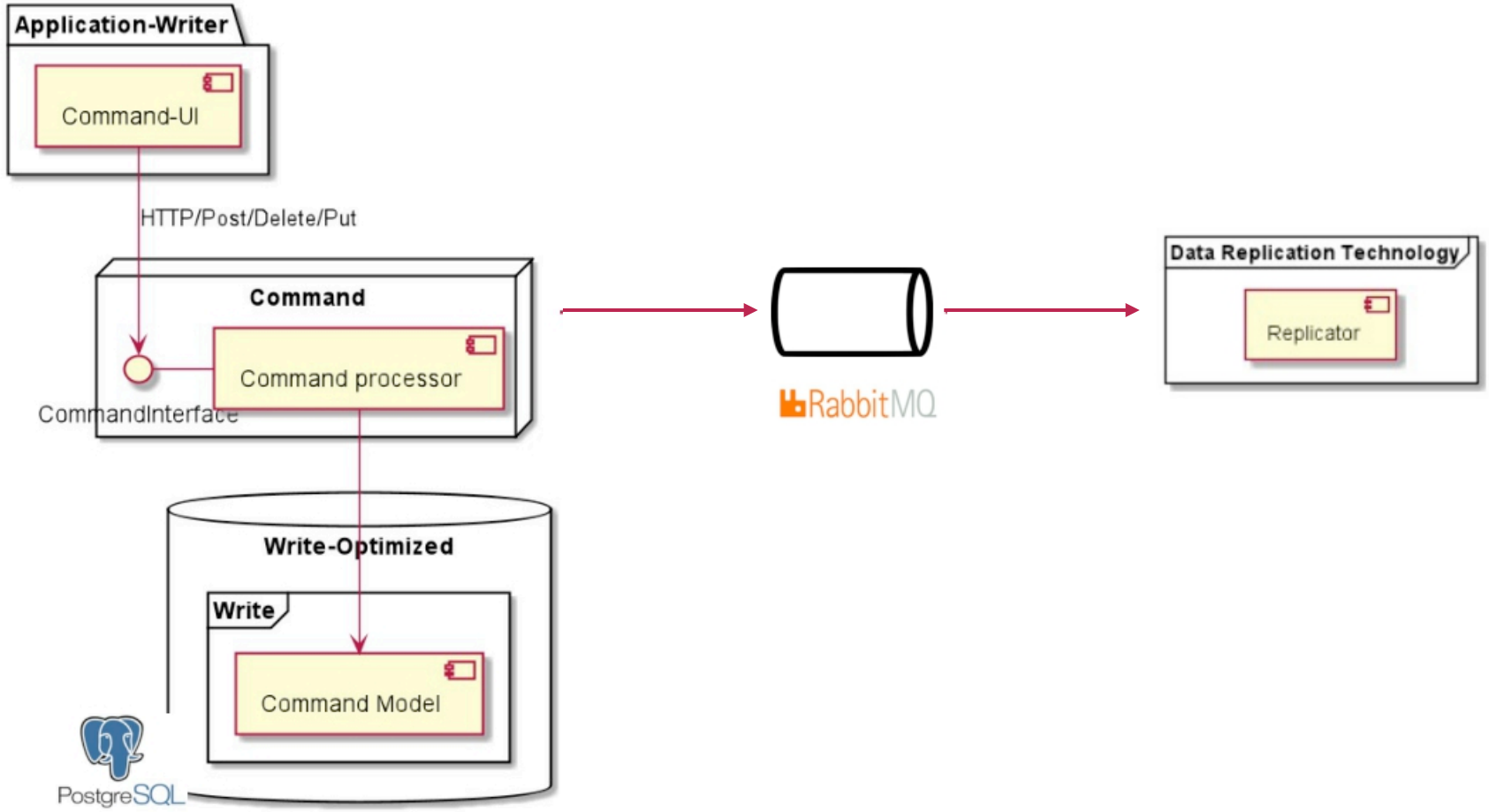
Always anticipate that there will be FAILURES

- Identify the "Failure Points" in your architecture



Pro - actively address the "Failure Points"

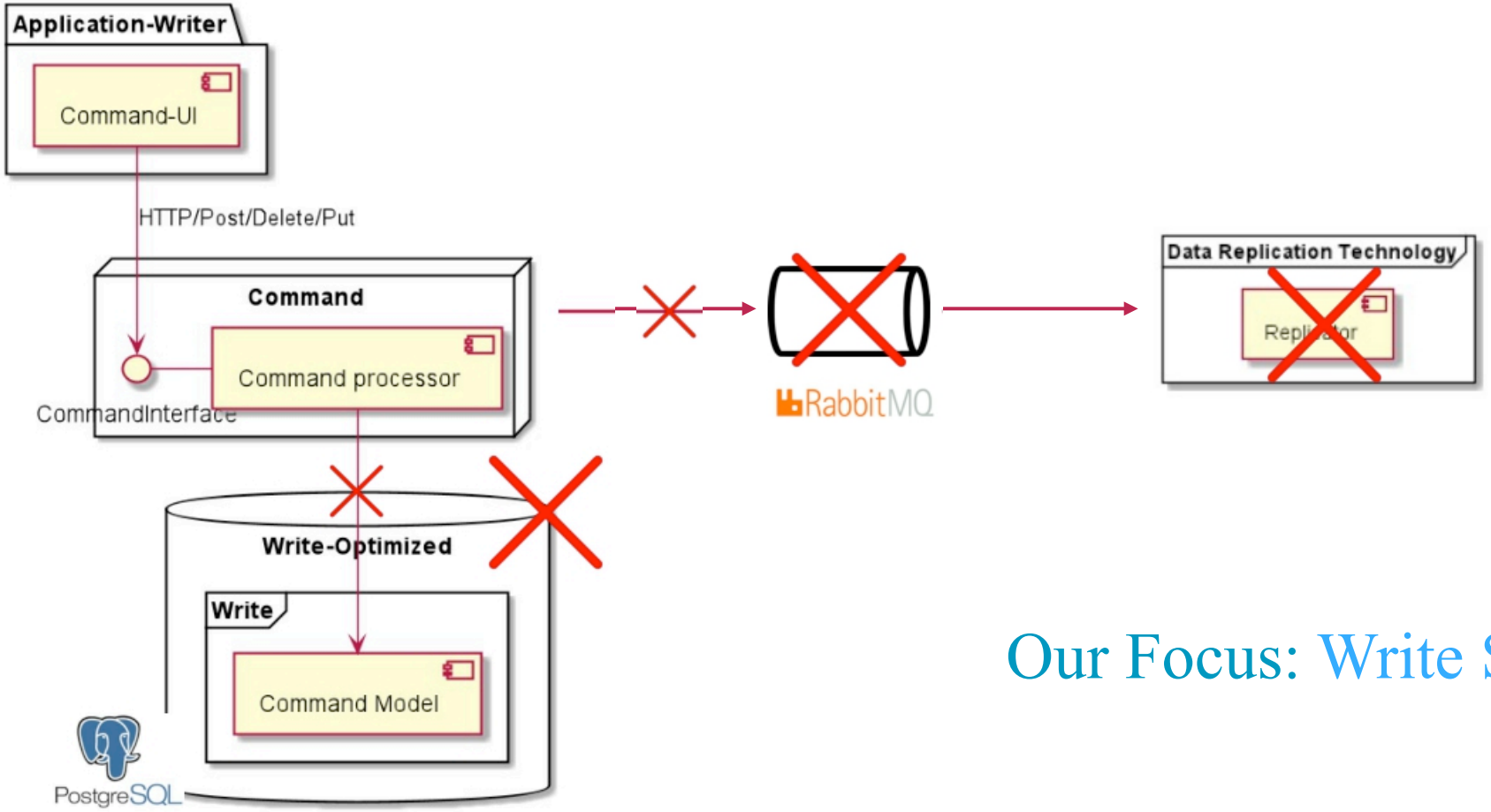
Command Writes to DB & MQ



Identify failure points in this?

Failures in Distributed systems

ASSUME that there will be failures in all interfaces | components

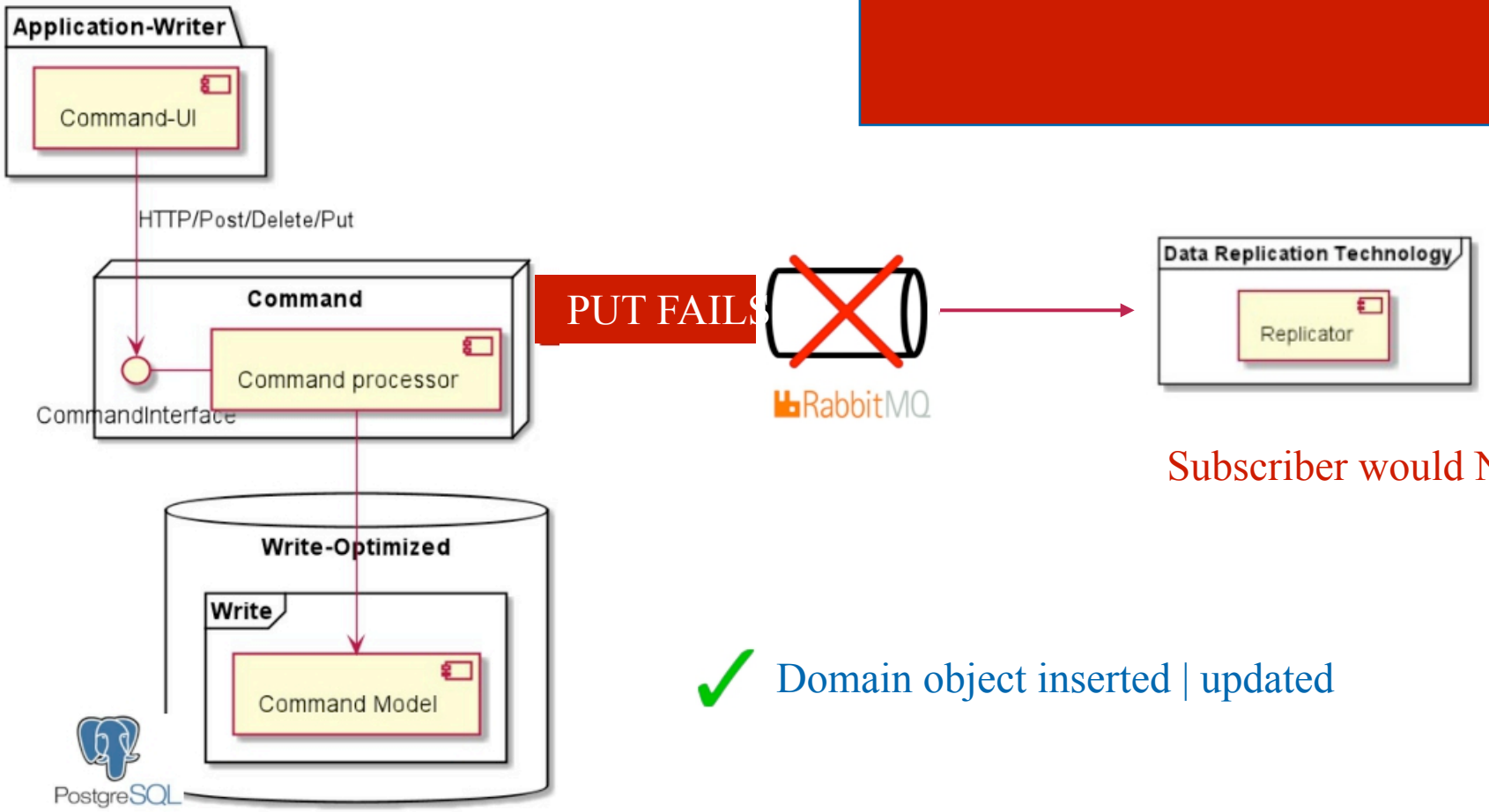


Our Focus: Write Side

Identify the impact

Impact of MQ Failure

WRITE & READ Side will be Out of Sync !!!



Subscriber would NOT receive event !!!

✓ Domain object inserted | updated

Solution options

DB Write & MQ Publish in a single unit of work (transaction)

2 Phase Commit

Distributed algorithm that coordinates all the processes involved in distributed transaction

- Also known as eXtended Architecture or XA
- Transaction manager coordinates with resources

https://en.wikipedia.org/wiki/Two - phase_commit_protocol

Solutions

DB Write & MQ Put in a single unit of work (transaction)

2 Phase Commit

Distributed algorithm that coordinates all the processes involved in distributed transaction

- Many commonly used distributed technologies do NOT support it !!!!

RabbitMQ

amazon SQS

mongoDB

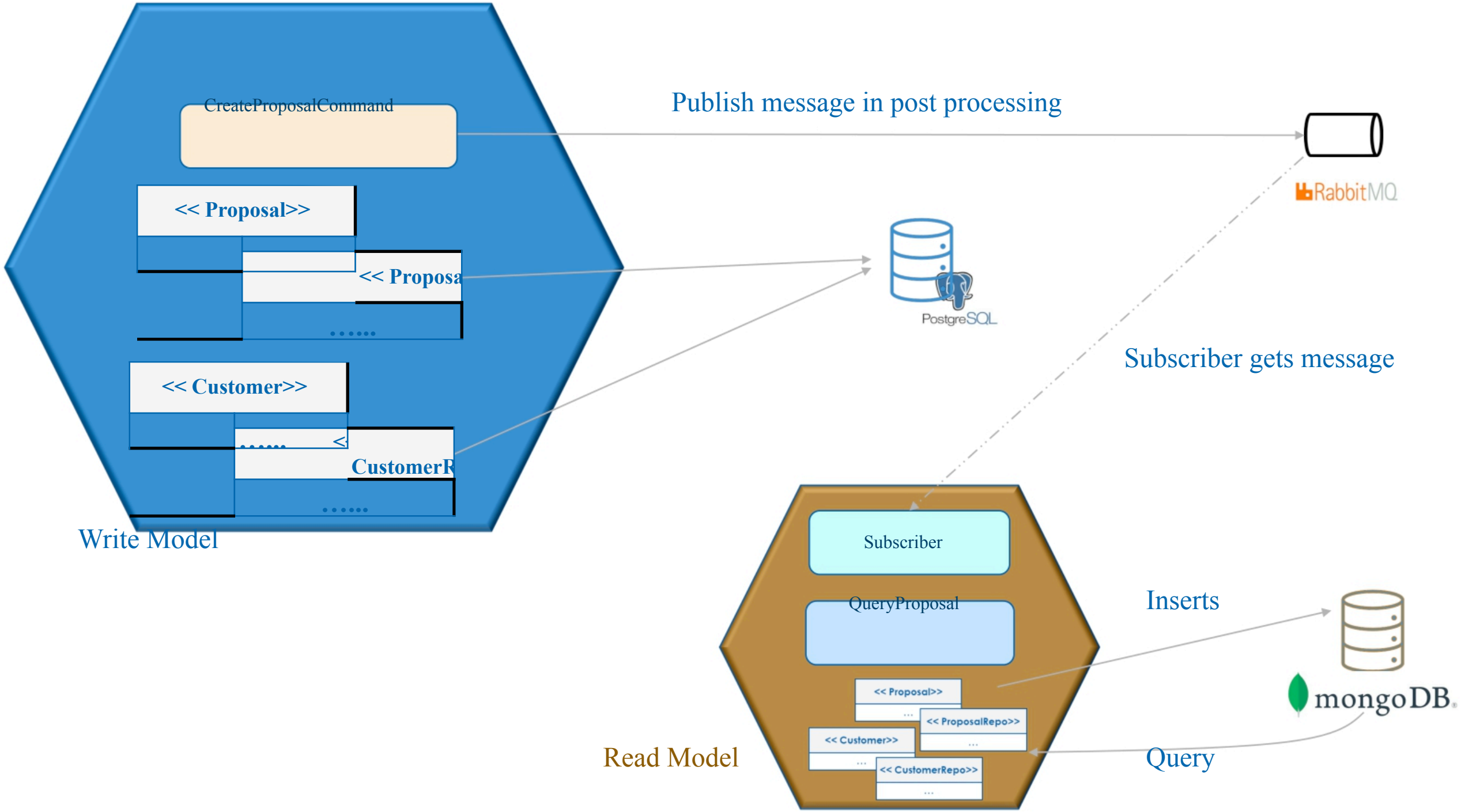
https://en.wikipedia.org/wiki/Two - phase_commit_protocol

Solutions

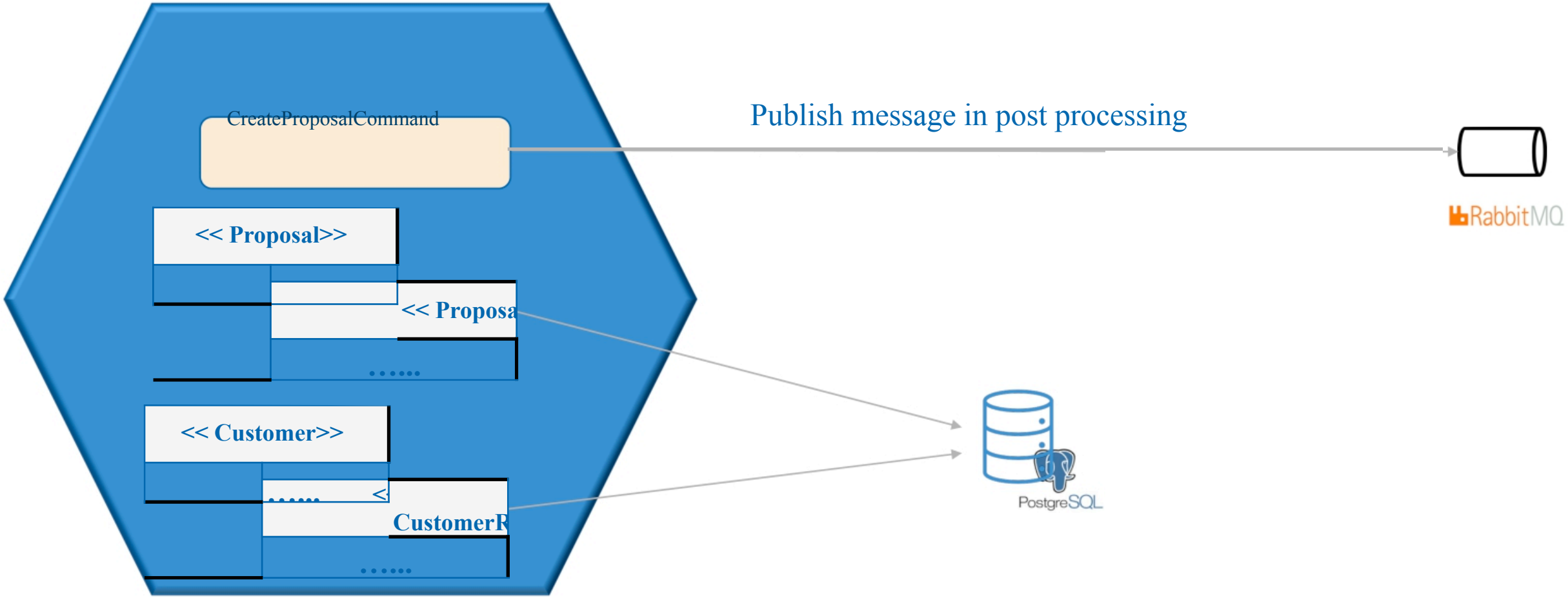
Break the DB Write & MQ Publish in two steps + Local Transaction

Reliable Messaging pattern

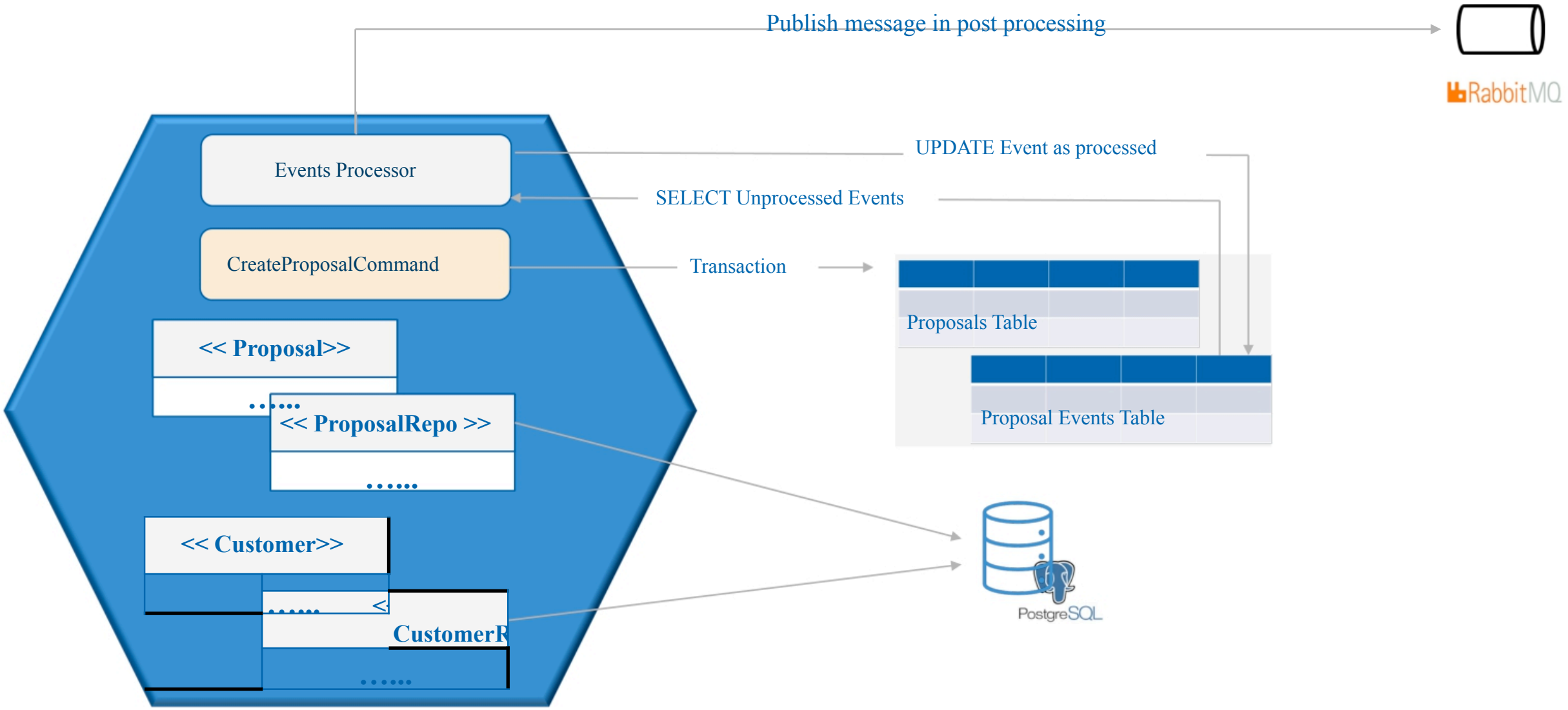
Write domain object data & event data in the database with a local transaction and replay the events against queueing system in a separate | subsequent step.


CreateProposalCommand
Publish message in post processing
RabbitMQ
<< Proposal>>
<< Proposa
.....
PostgreSQL
Subscriber gets message
<< Customer>>
.....
CustomerR
.....
Write Model
Subscriber
QueryProposal
Inserts
mongoDB
<< Proposal>>
...
<< ProposalRepo>>
...
<< Customer>>
...
<< CustomerRepo>>
...
Read Model
Query

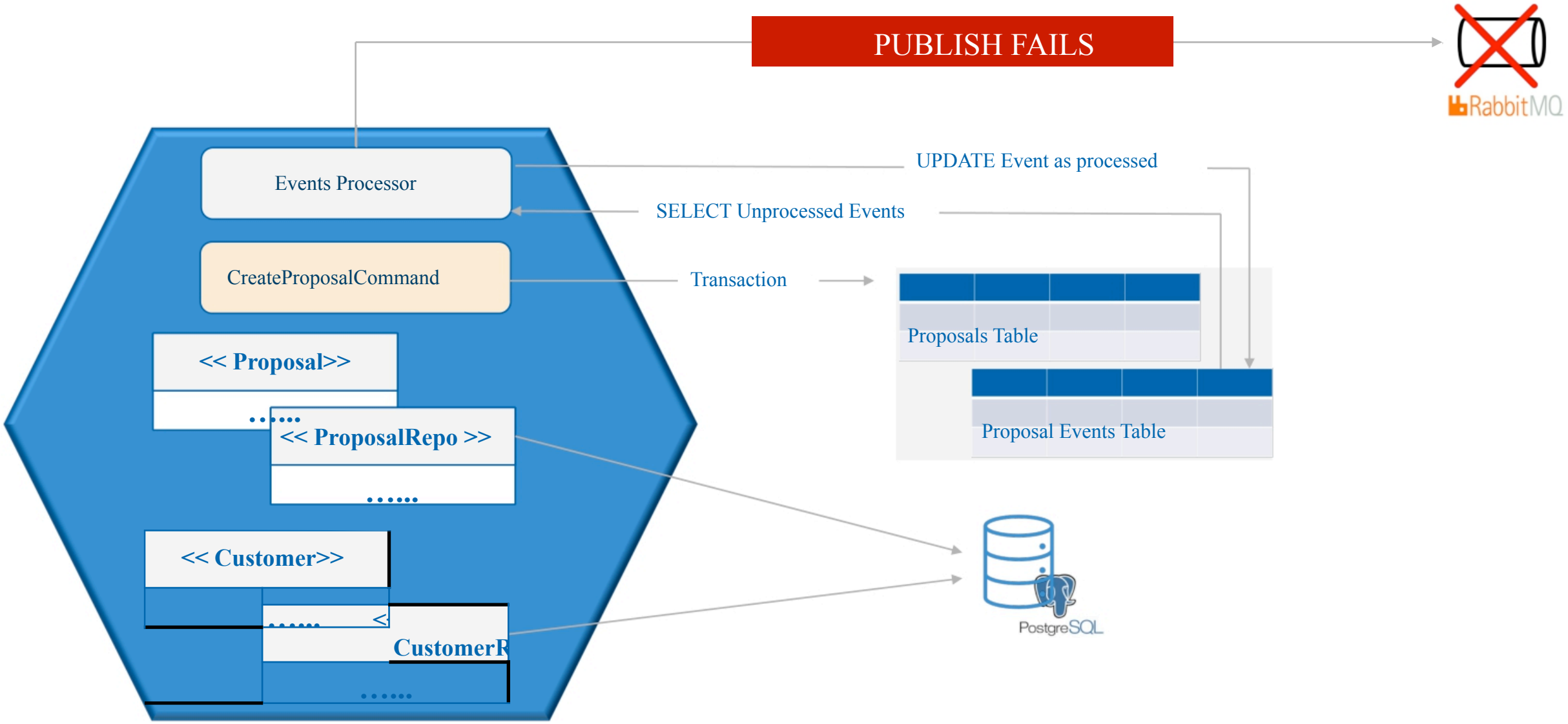
Write side messaging



Write side messaging

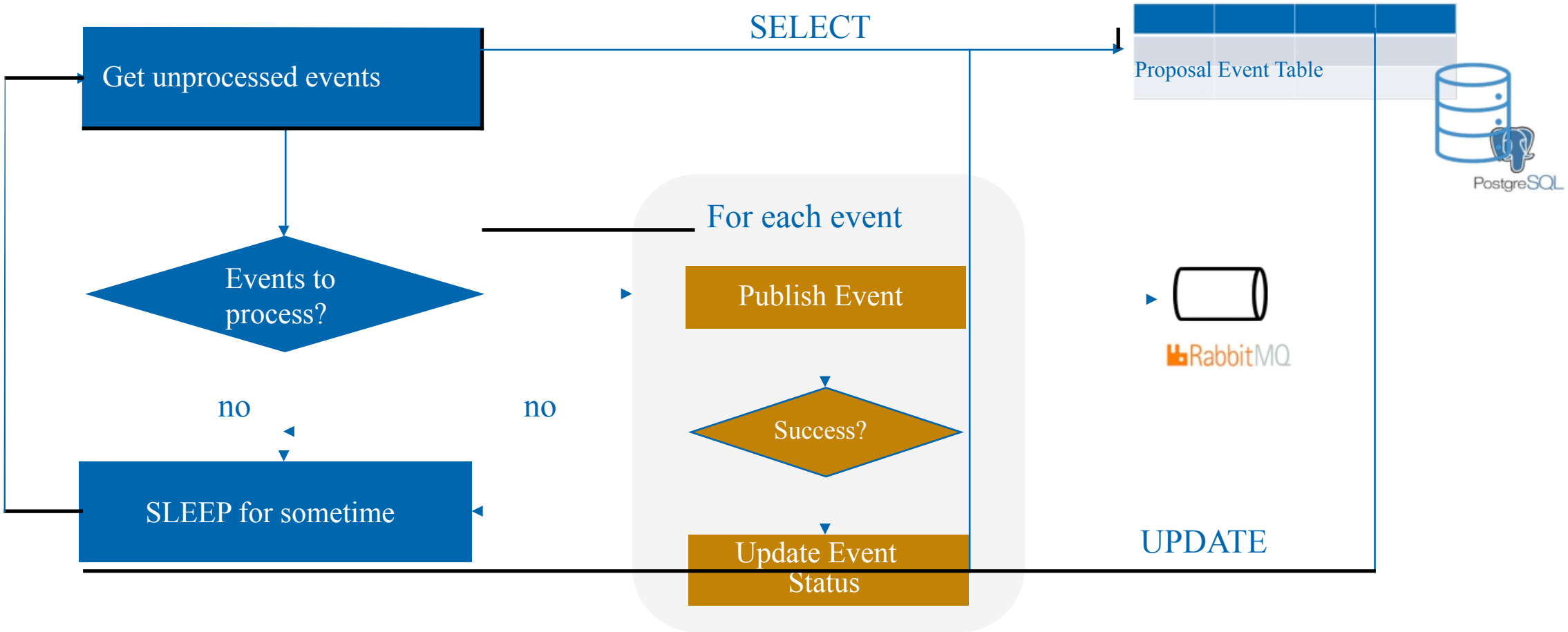


Write side messaging



Event Processor Logic

Independent | Standalone process

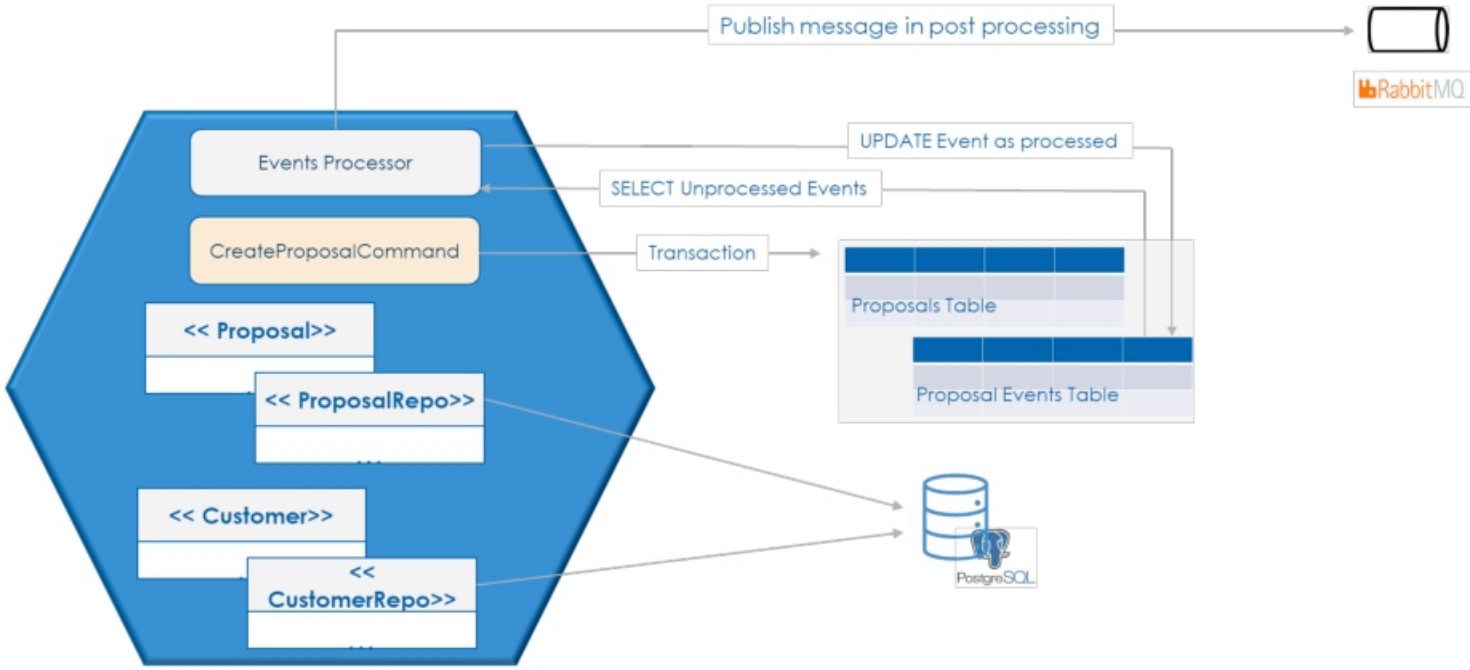


Quick Review

MUST identify failure points & address

2 Phase Commit has multiple challenges

Reliable Messaging Pattern



Write Side Update

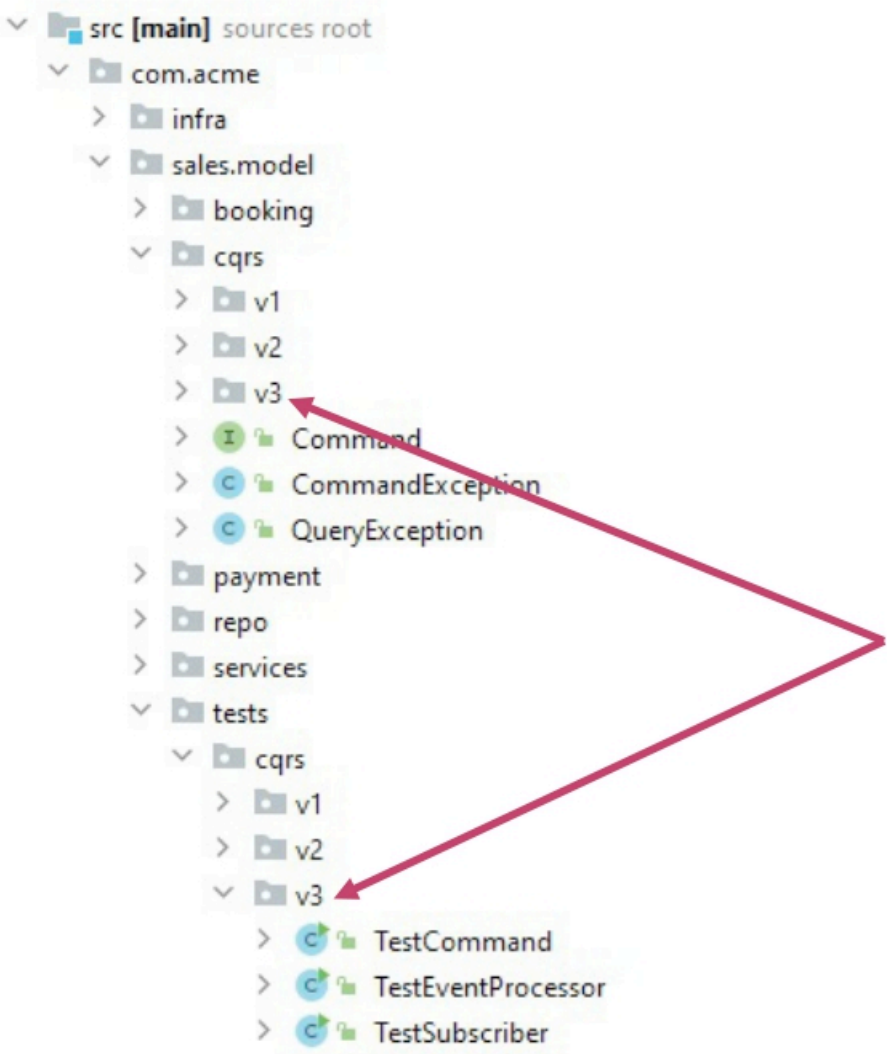
Enhance CQRS Write to prevent event message loss

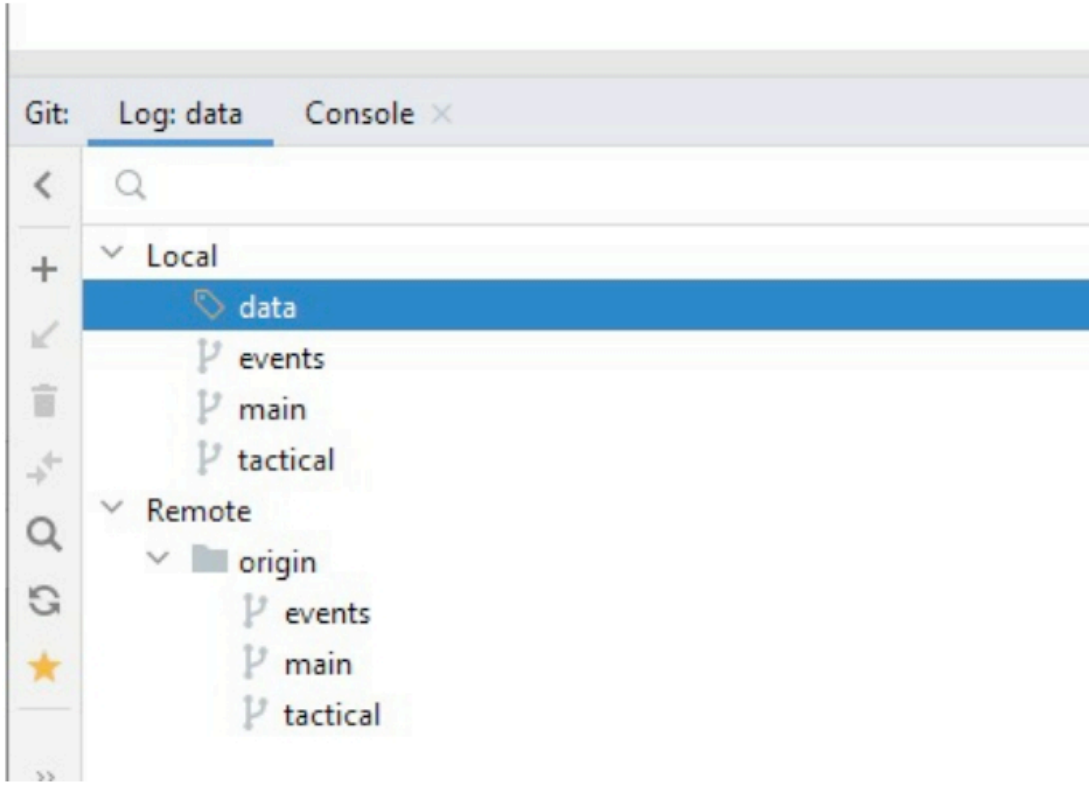
1. Walkthrough of Implementation
2. Walkthrough of Code - EventProcessor
3. Test the code

Code

Checkout the branch : data

- Instructions in README.md





IT Lead

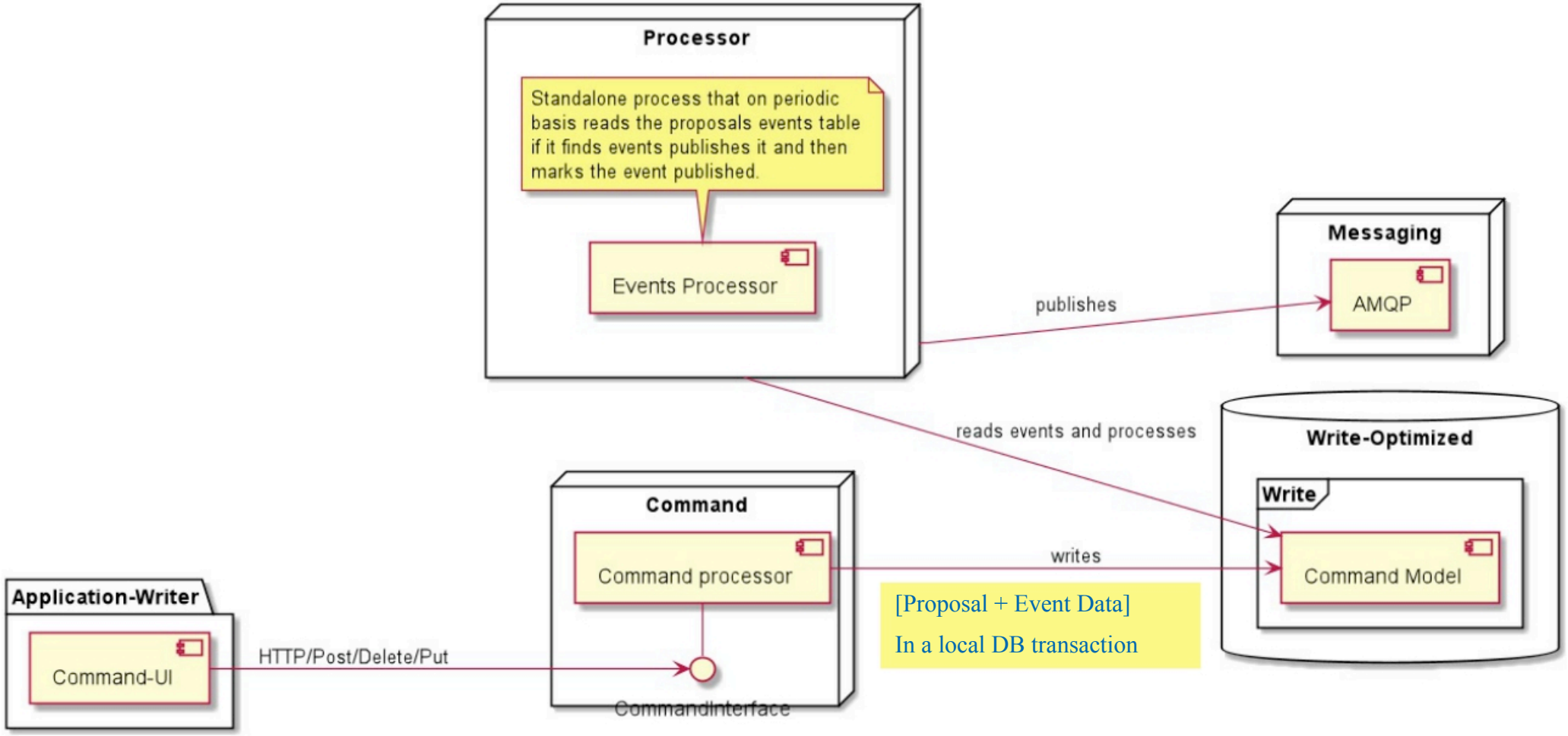
Proposal Events CQRS Testing

Events are LOST when Rabbit MQ is down!!

Build Reliable Messaging to fix it !!!!

Write side Components

uml/ cqrs/cqrs.writeside.v3.component.puml



Setup the DB tables : proposal events table

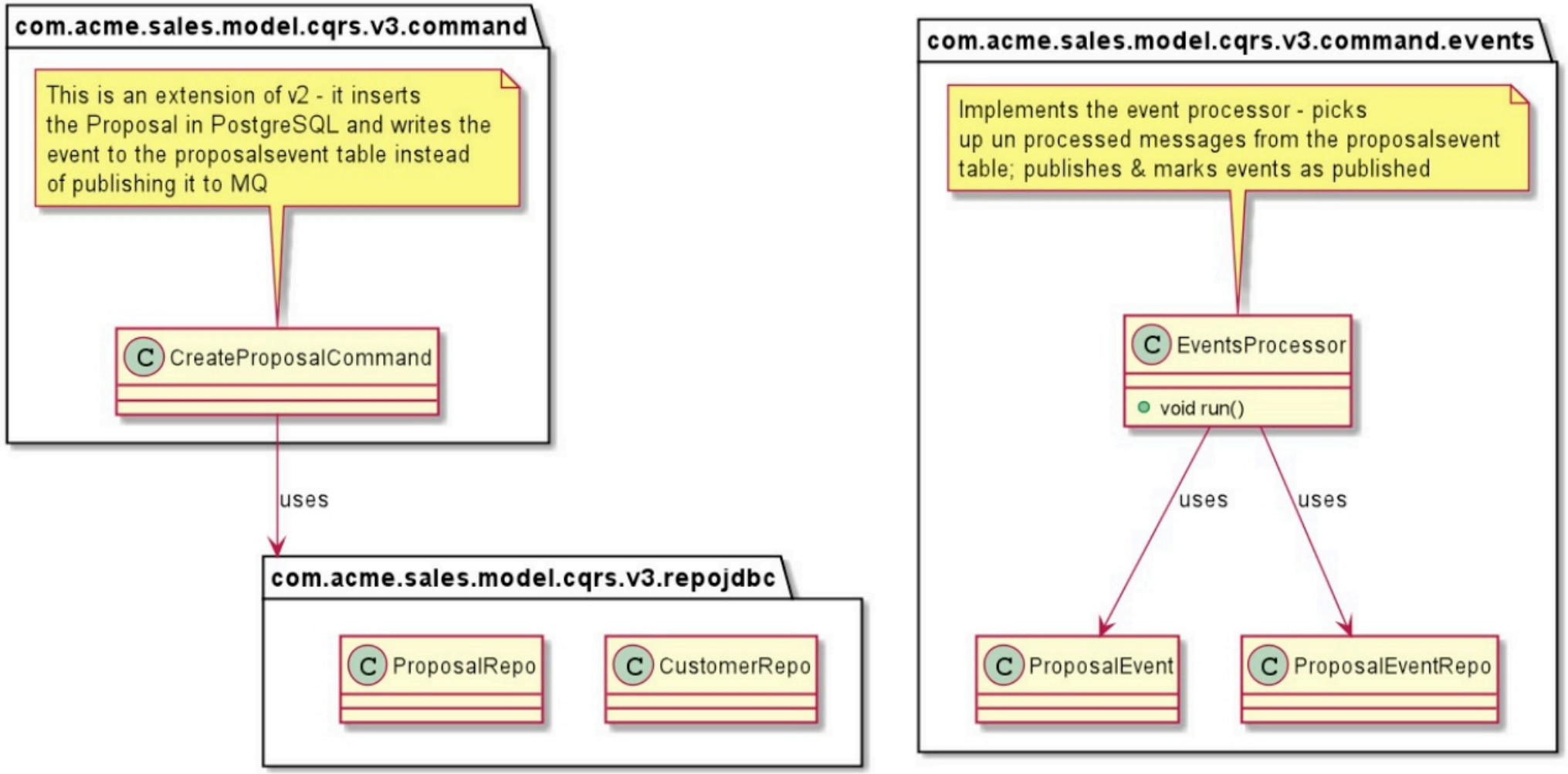
sql/reliable -messaging -ddl.sql

```sql
CREATE TABLE proposalevents (
    event_id      INT  generated by default as identity,
    proposal_id   INT,

    event_guid    VARCHAR(36) not null,

    payload       VARCHAR(1024) not null,

    processed     BOOLEAN DEFAULT FALSE,

    created_date  TIMESTAMP DEFAULT now(),
    processed_date  TIMESTAMP,

    PRIMARY   KEY(event_id),

    CONSTRAINT fk_customer
        FOREIGN KEY(proposal_id)
            REFERENCES proposals(proposal_id)
);
```

PostgreSQL

Event processing classes

uml/ cqrs/cqrs.writeside.v3.class.puml



READ Side Failures

Addressing READ side failures

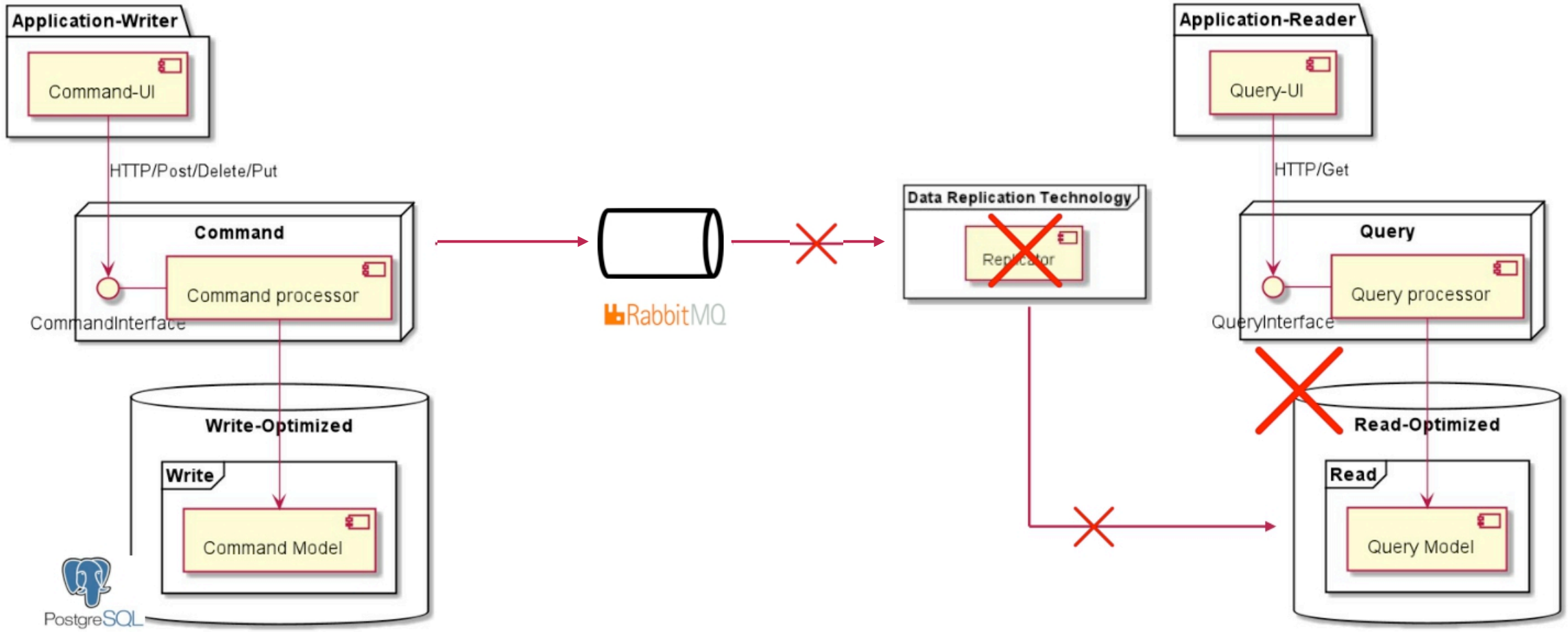
1. Read side failure scenarios
2. Message duplication
3. Solution

PLEASE NOTE:

Discussion is for messaging - based replication

Does NOT apply to Kafka Streams

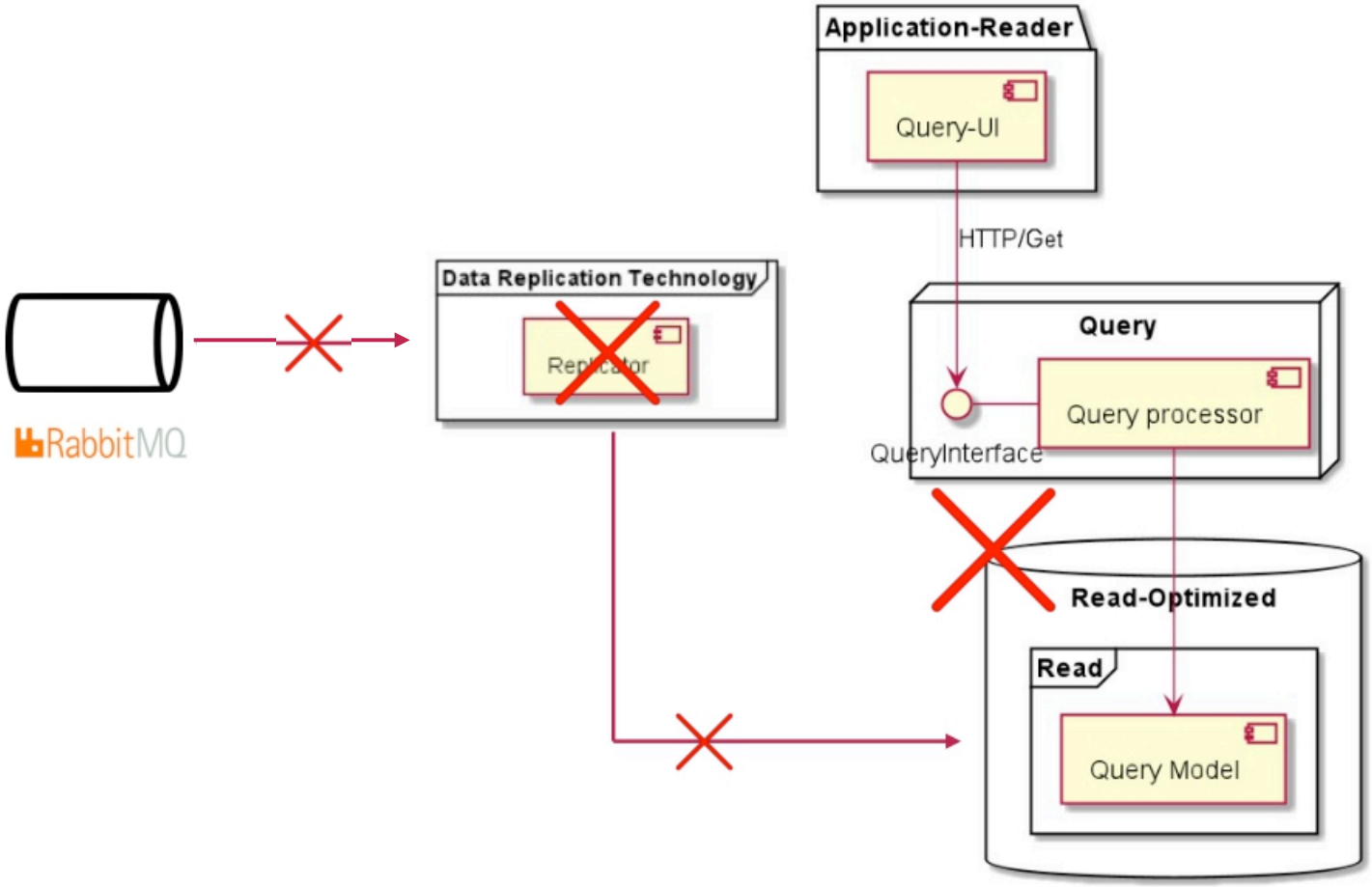
Identify Failure points on READ side?



Impact of Read side Failures

Loss of events => READ & WRITE side out of Sync

Longer time to consistency



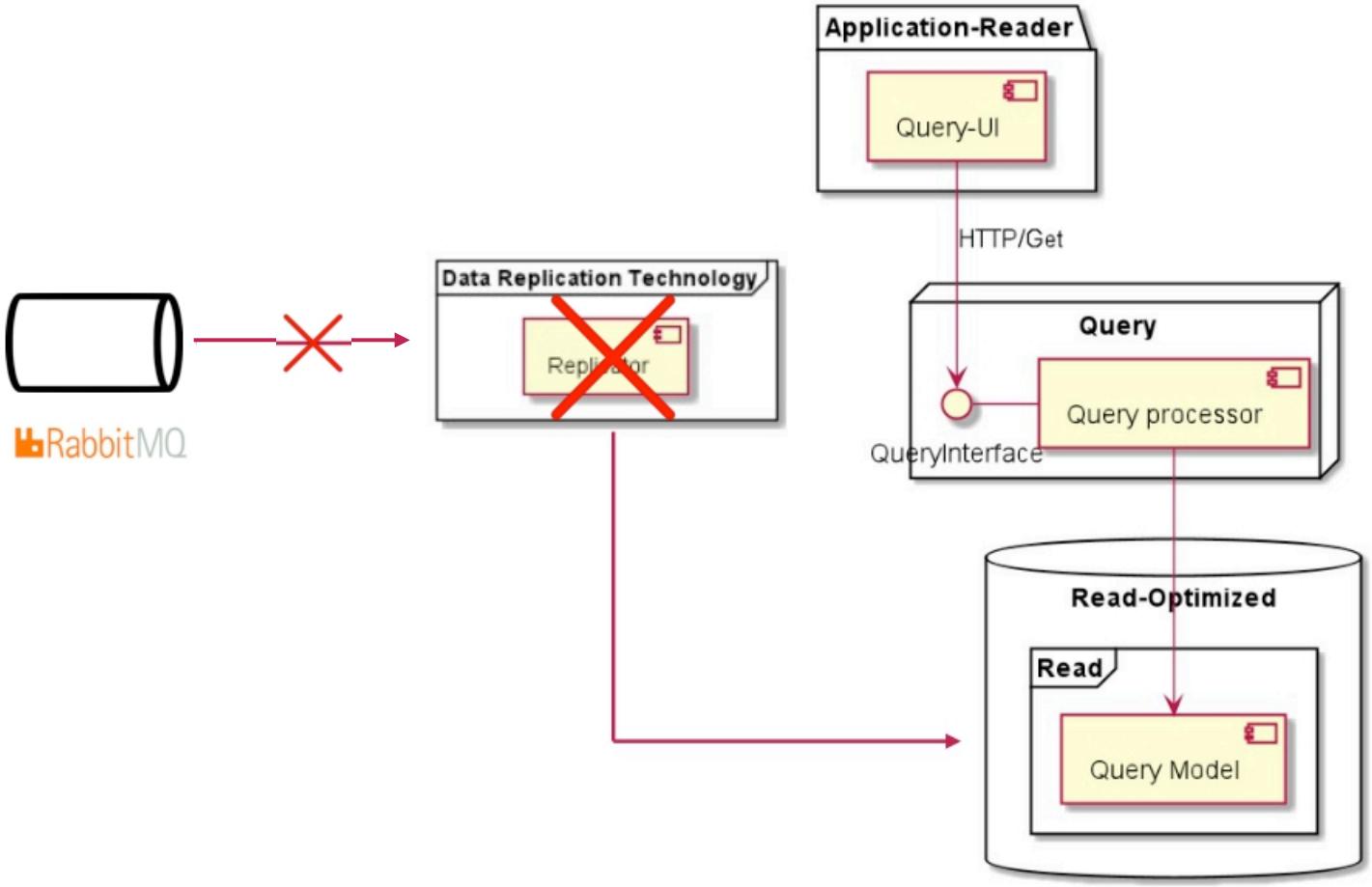
Solution: Replicator could not reach MQ or failed to process

Use message Persistence & TTL on MQ Bus

Message stays in the Queue for a set time after which the message is lost

Most MQ bus allow setup of TTL for the Queue and at message level

https://www.rabbitmq.com/ttl.html

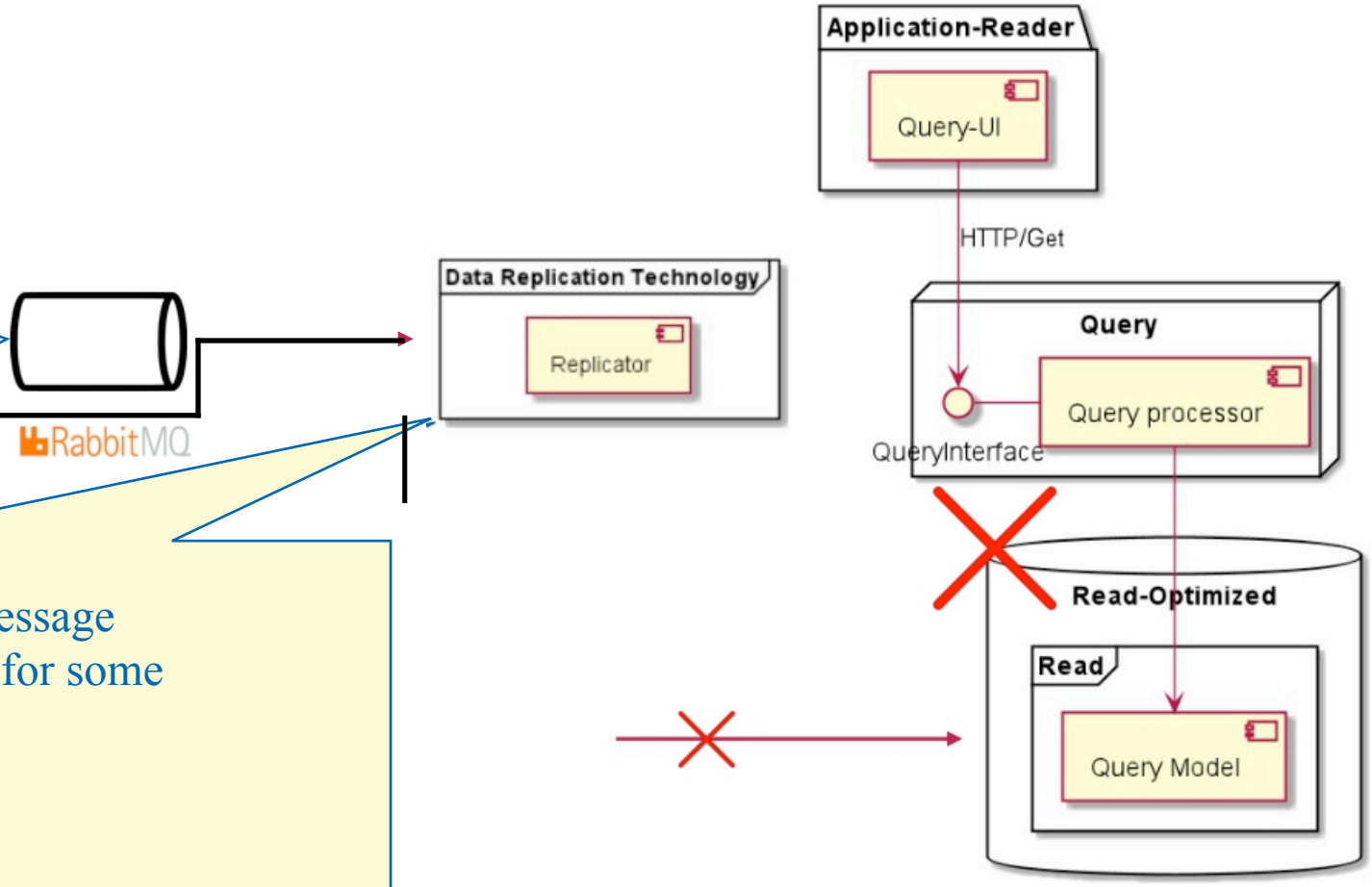


Solution : Replicator failed to update Read side DB

Use reject | rollback to prevent message loss

- Message may end up in Dead Letter Queue after certain number of retries
- Queue length may exceed, and message will be lost

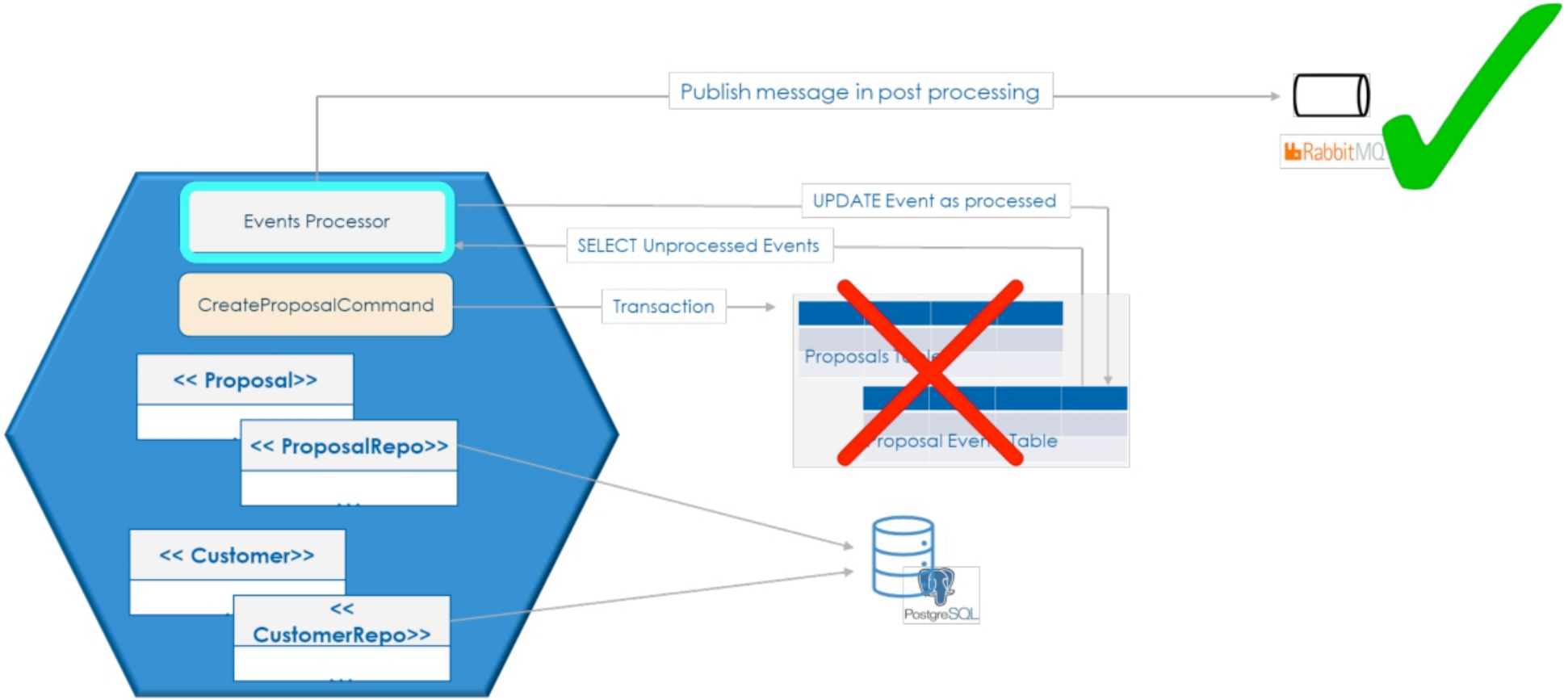
https://www.rabbitmq.com/dlx.html



Message subscriber on failure can reject the message which is like a rollback; Subscriber can then wait for some time before retrying

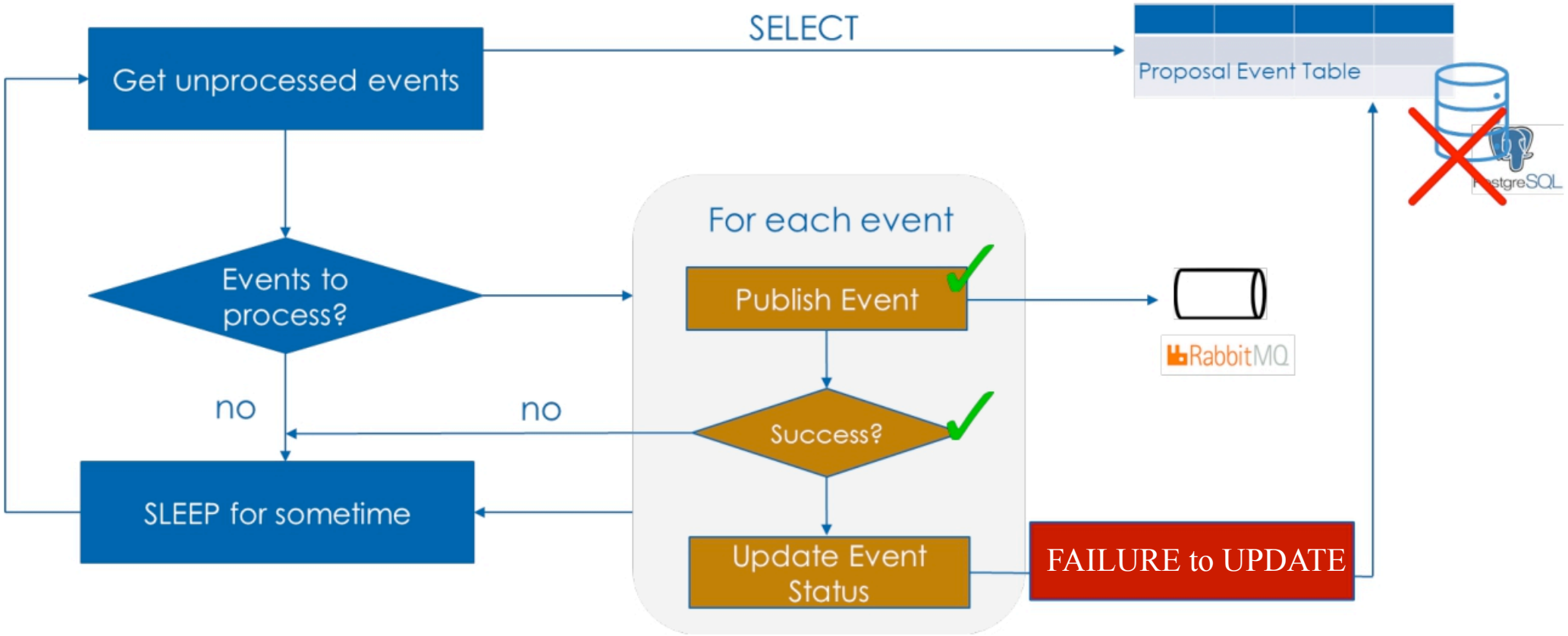
https://www.rbbitmq.com/semantics.html

WRITE Side failure's impact on READ side !!!



What will be the Impact of this failure on READ side?

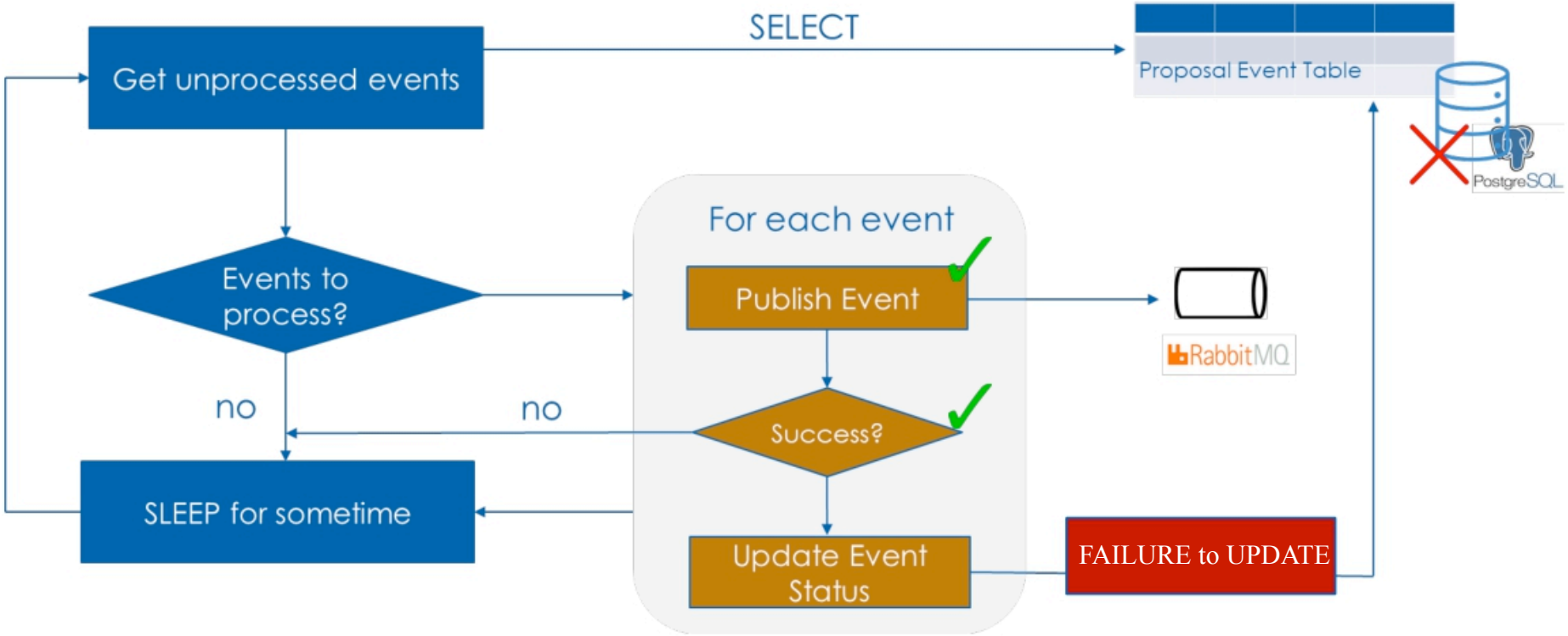
WRITE Side failure's impact on READ side !!!



Impat of this failure on READ side?

Event Processor Logic

What if the Database crashes before status update?

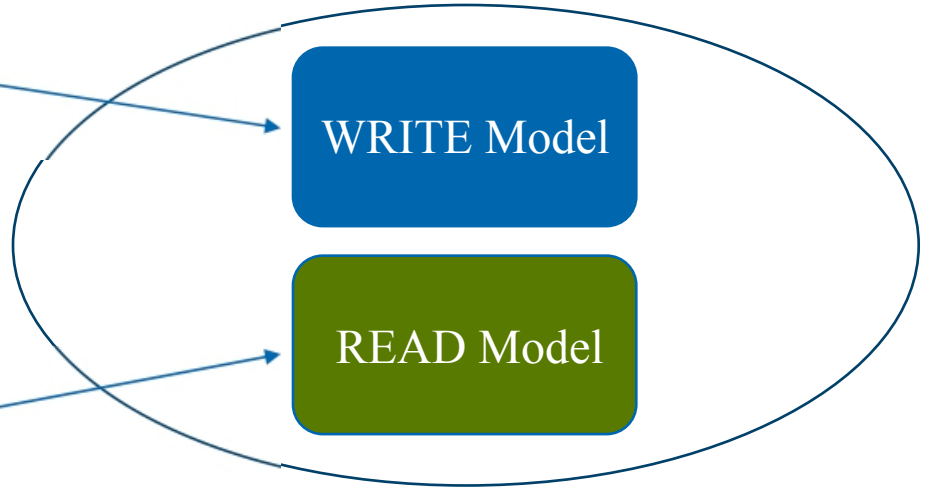


Duplication of Events | Messages on the READ Side !!!

Which side should solve this issue - READ or WRITE?

"Design for Failure !!!"

Can't depend on other teams
to solve our problem ☺



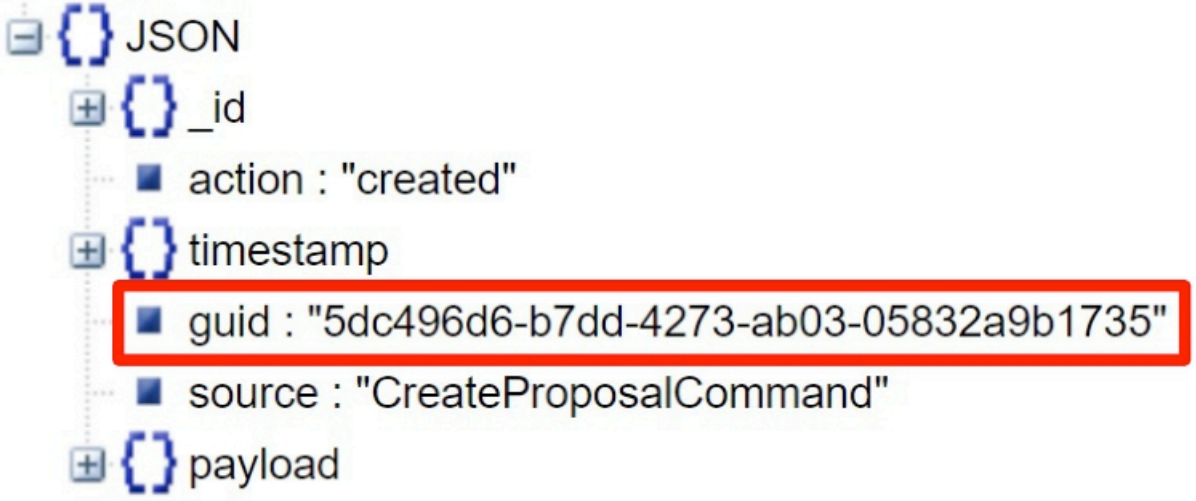
BOTH sides should consider putting in a fix !!!

Solution

Duplicate events may be ignored

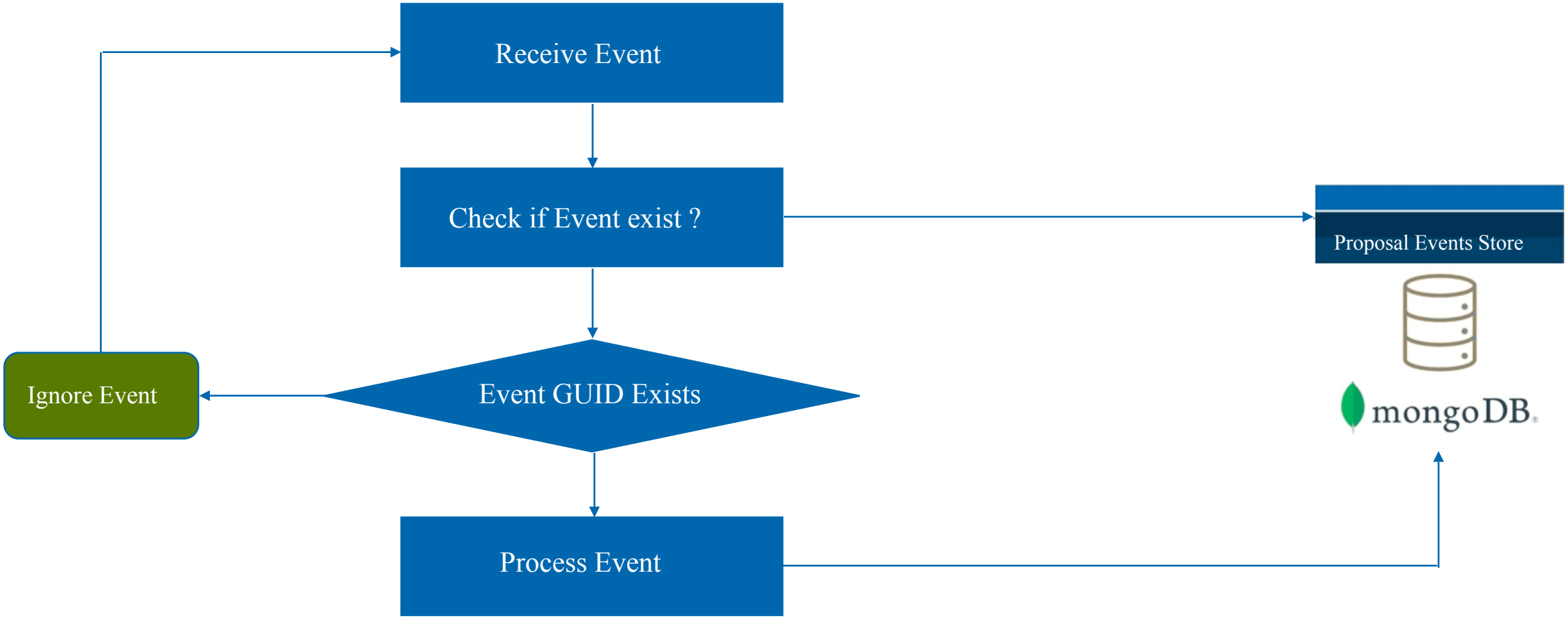
- OR logged in a separate DB for investigation

- Each event needs UNIQUE identity



Read side Solution (Subscriber)

Check if event has already been processed



Quick Review

Some types of failures may impact downstream systems

Each team MUST "Designs for Failure"

Read Side Update

Prevent duplicate messages

1. Class & Sequence diagram
2. Test the code

IT Lead

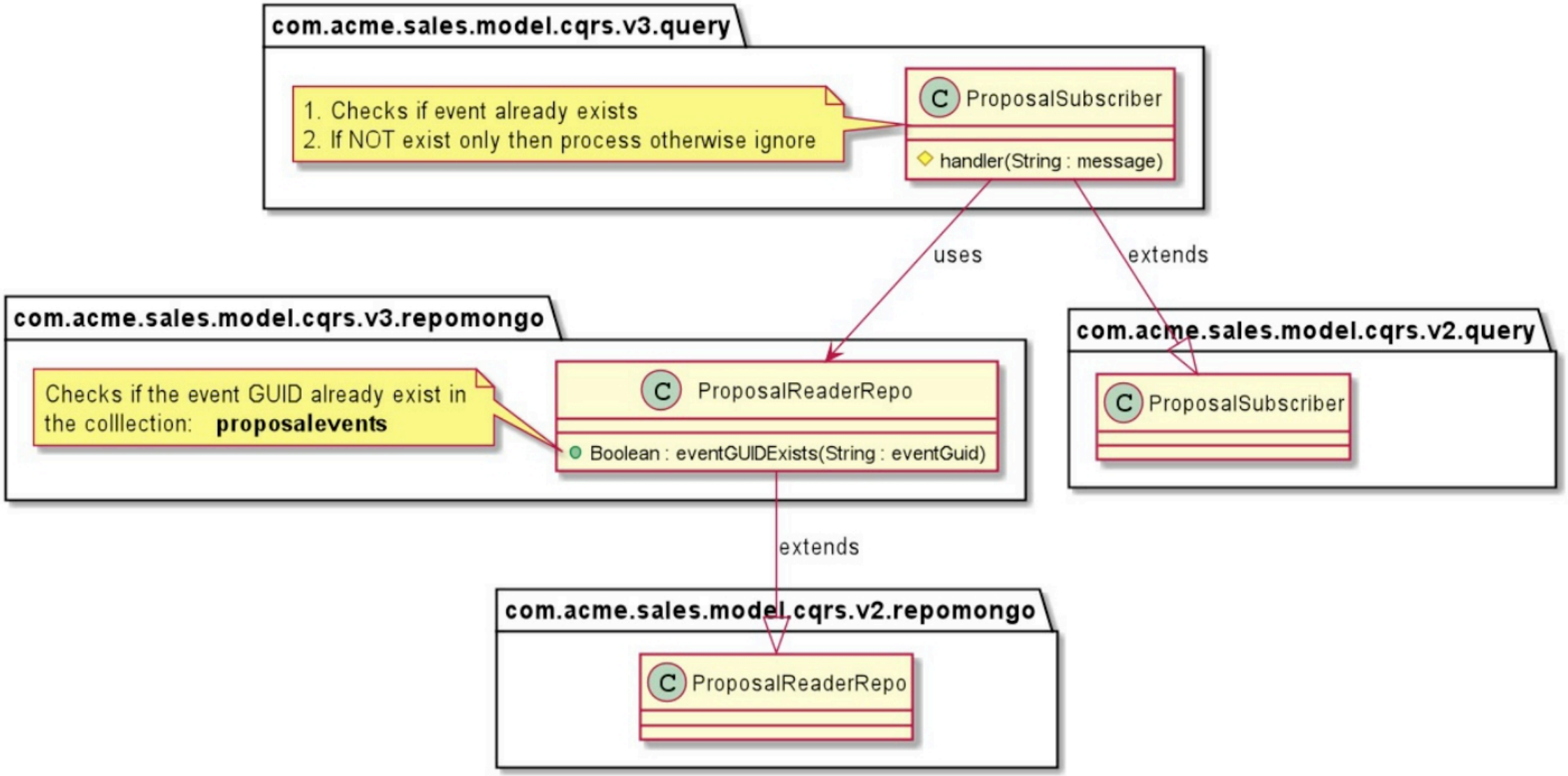
Proposal Events CQRS Testing

Events are getting duplicated !!!

Fix the READ side to address the issue !!!

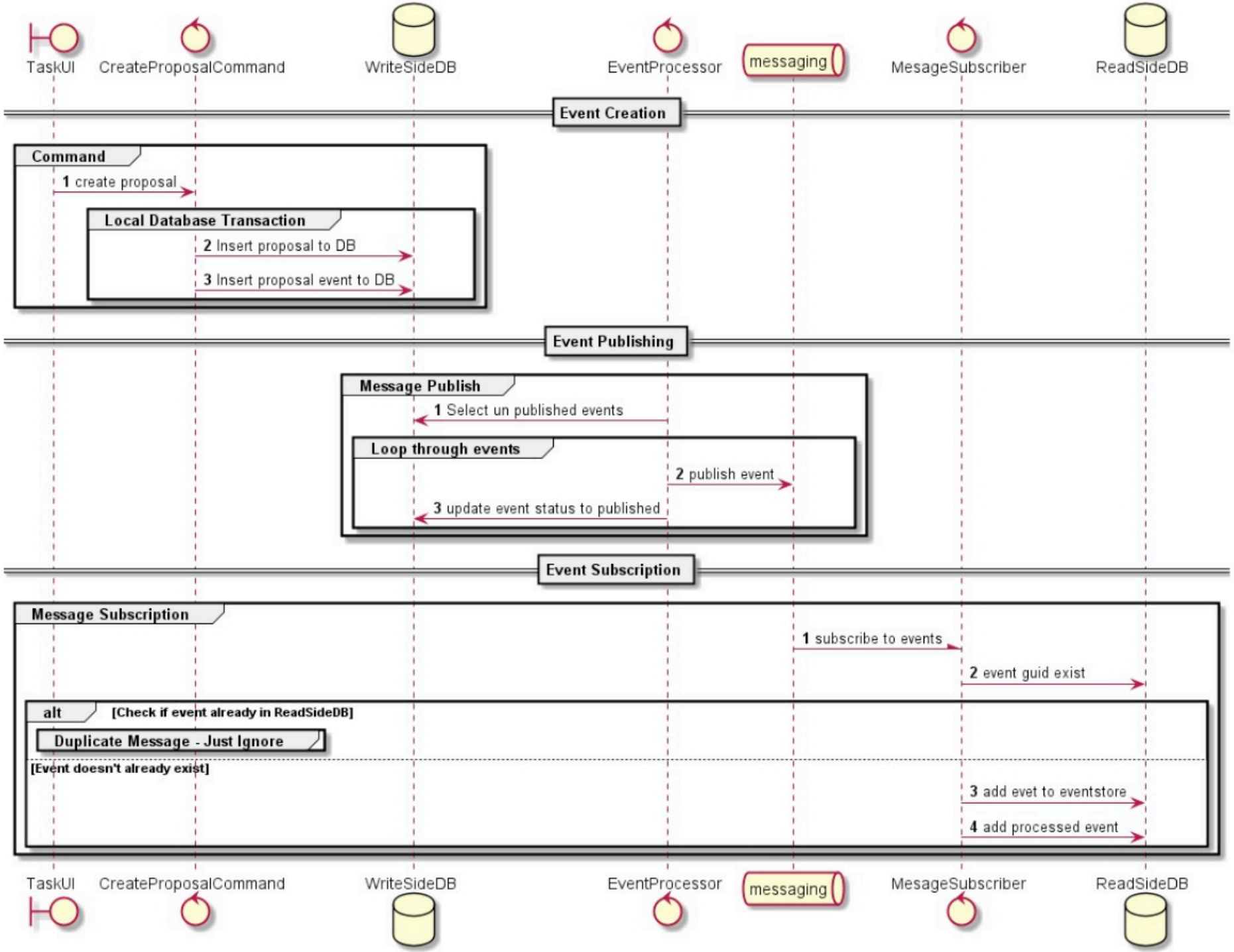
Read side classes

uml/ cqrs/cqrs.readside.v3.class.puml



Message Flow

uml/cqrs/cqrs.v3.messageflow.sequence.puml



Testing - Cleanup

- Cleanup the proposalevents table

- Delete the proposal collections

Testing

1. Launch event processor
2. Launch the event subscriber
3. Execute test command
4. Send duplicate event by changing status of event in Write Side DB