



TRANSACTIONAL SOLUTION FOR MICROSERVICES

ADAM RŮŽIČKA A ONDŘEJ CHALOUPKA

ABOUT US

ADAM RŮŽIČKA

- Red Hat
- developer of Foreman project
- mainly Ruby
- github: github.com/adamruzicka

ONDŘEJ CHALOUPKA

- Red Hat
- developer at WildFly project
- mainly Java
- working on Narayana transaction manager
- github: github.com/ochaloup
- twitter: @_chalda

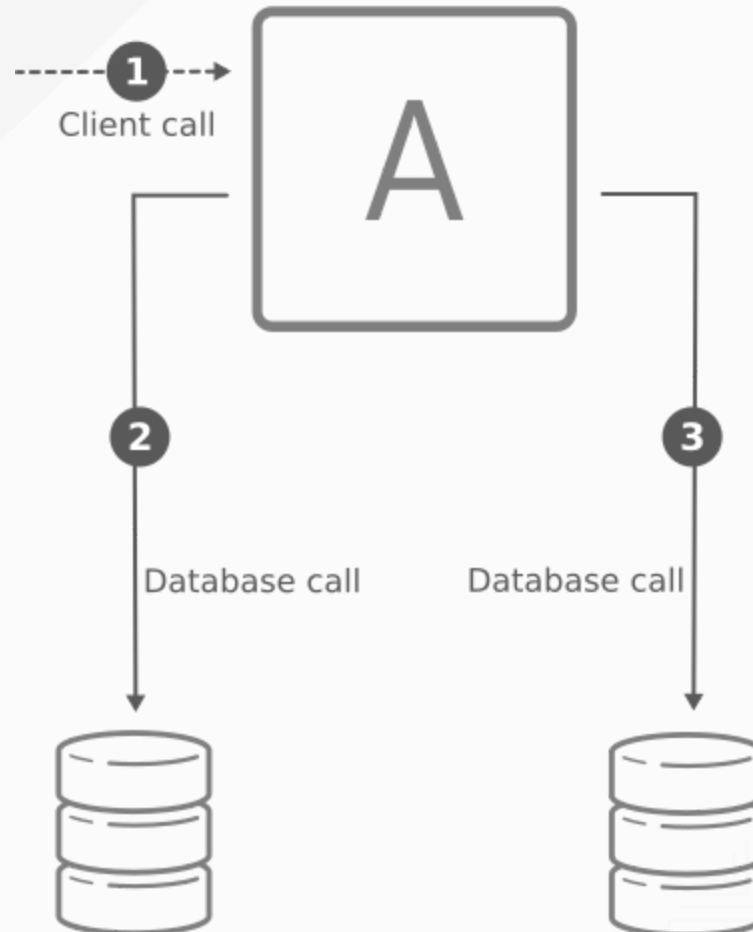
AGENDA

- what is transaction management
- microservices and transactions
- introduction to saga pattern
- Long Running Actions for MicroProfile
- DynFlow framework

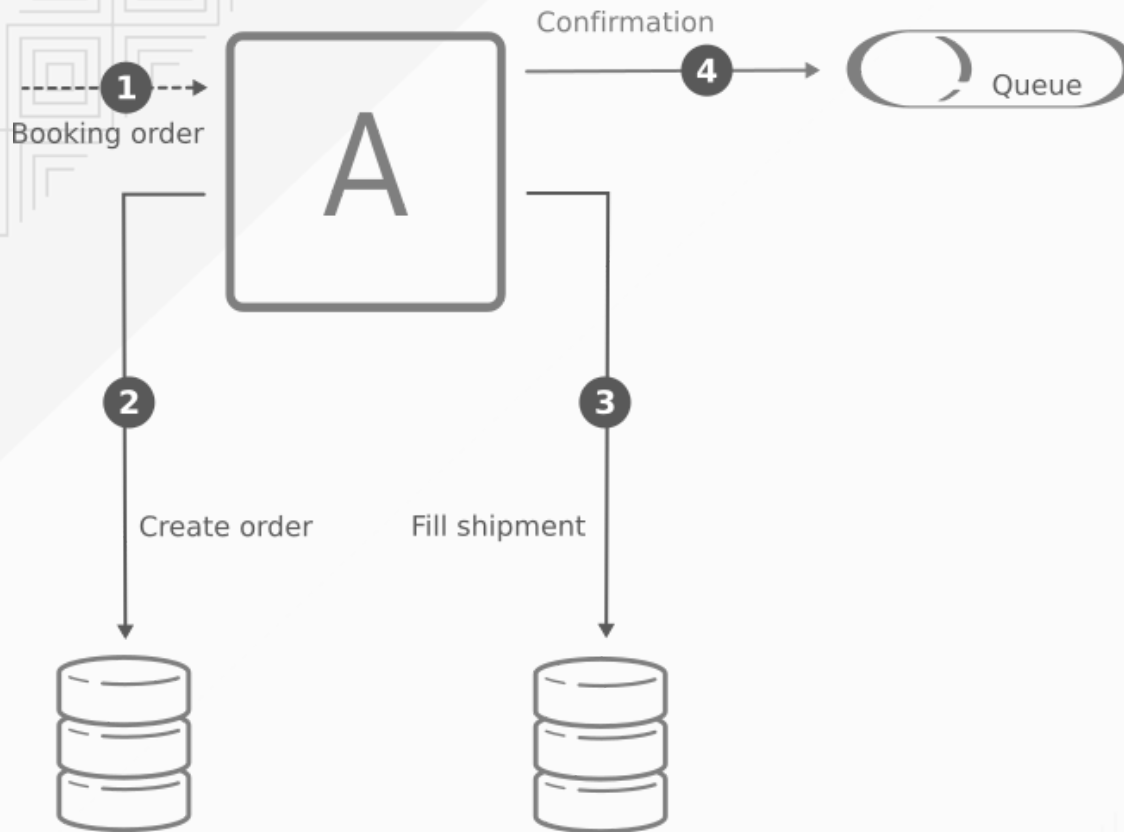
TRANSACTION

An atomic unit of work where all parts either finish with success or fail.

MONOLITHIC APPLICATION

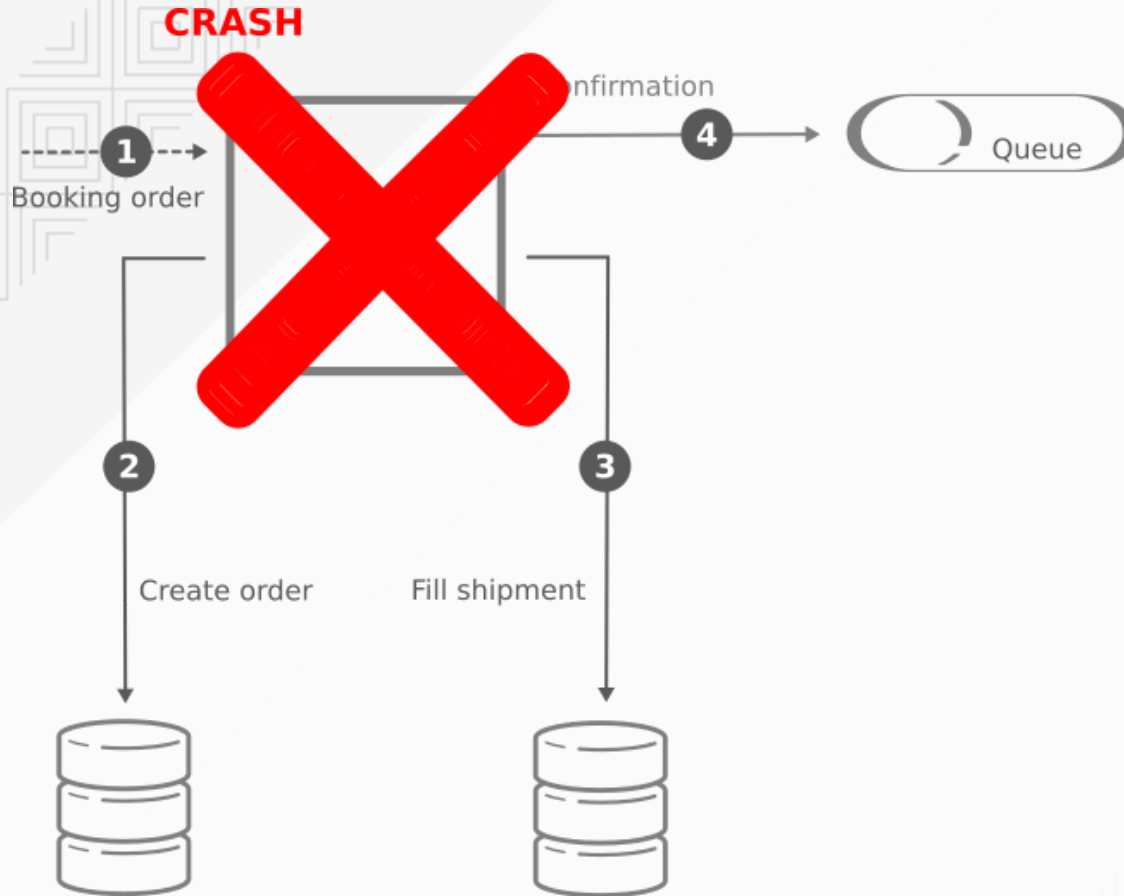


LET'S CREATE A BOOKING



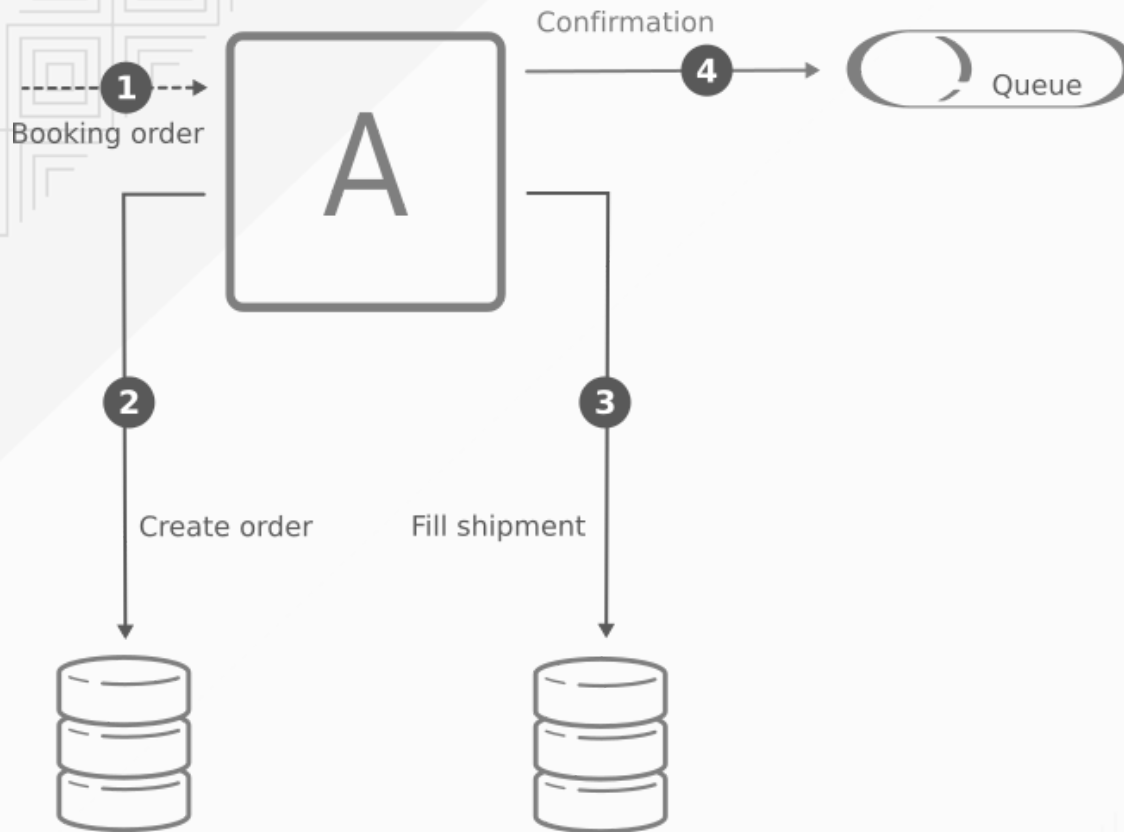
- Creating order
- Filling shipment
- Sending confirmation

...AND NOW WHAT?



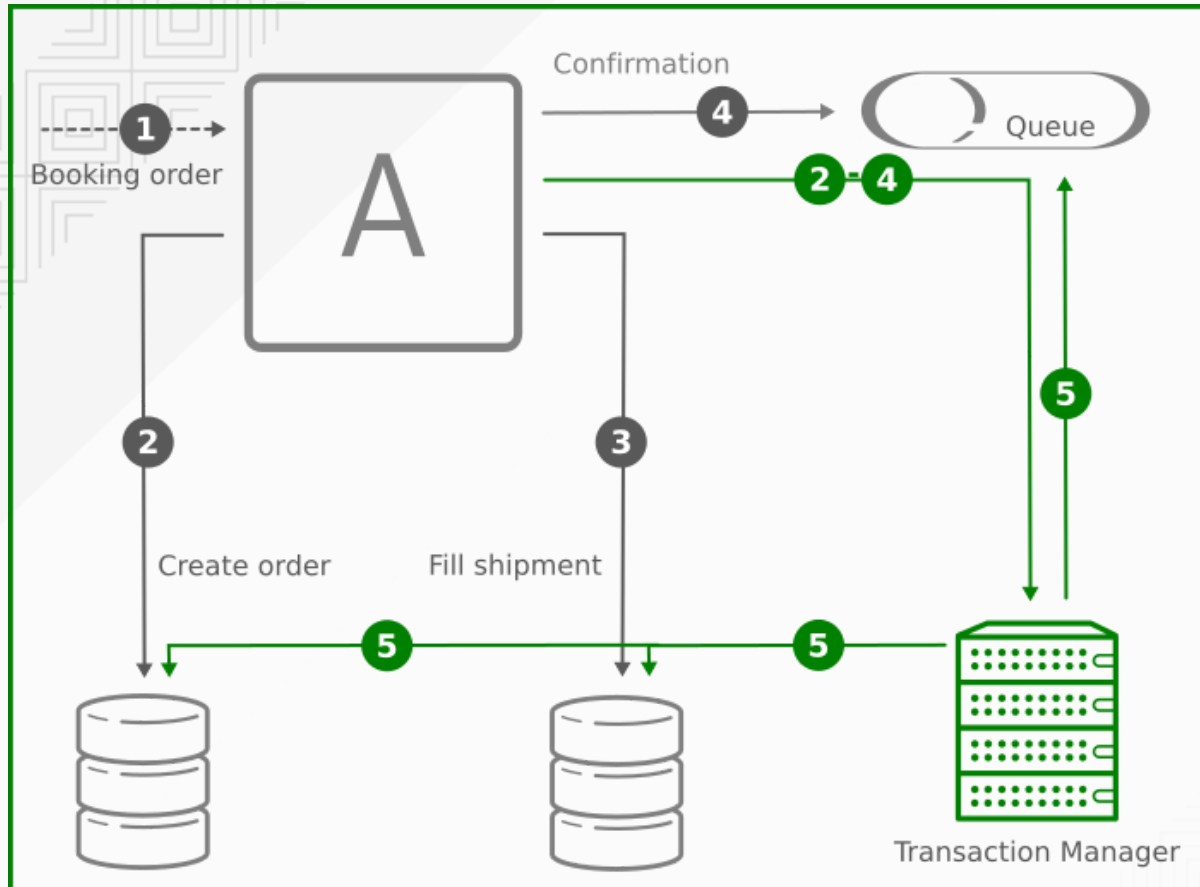
- Creating order
- Filling shipment
- Sending confirmation

A SINGLE UNIT OF WORK



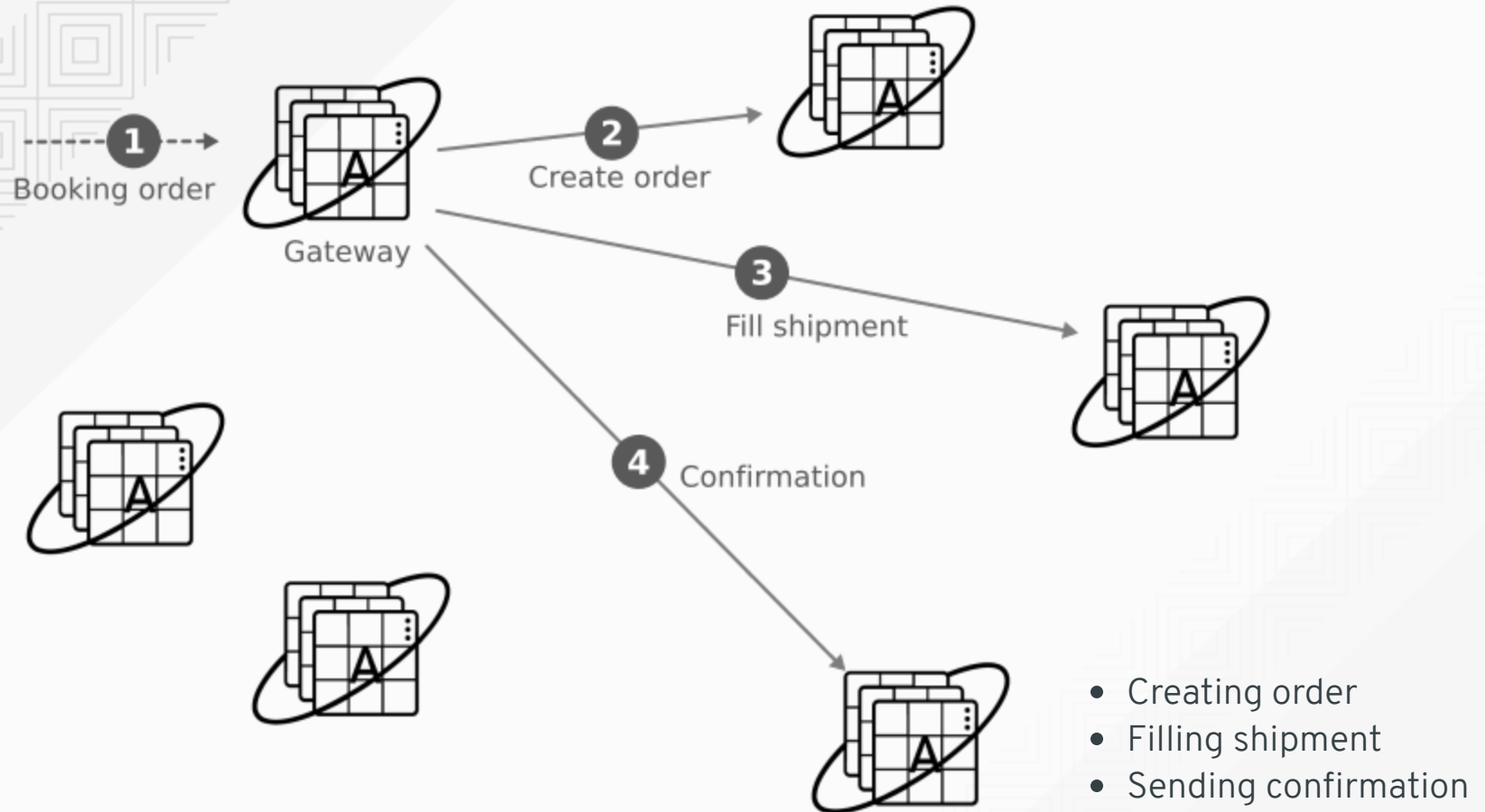
- Creating order
- Filling shipment
- Sending confirmation

ACID TRANSACTIONS TO RESCUE

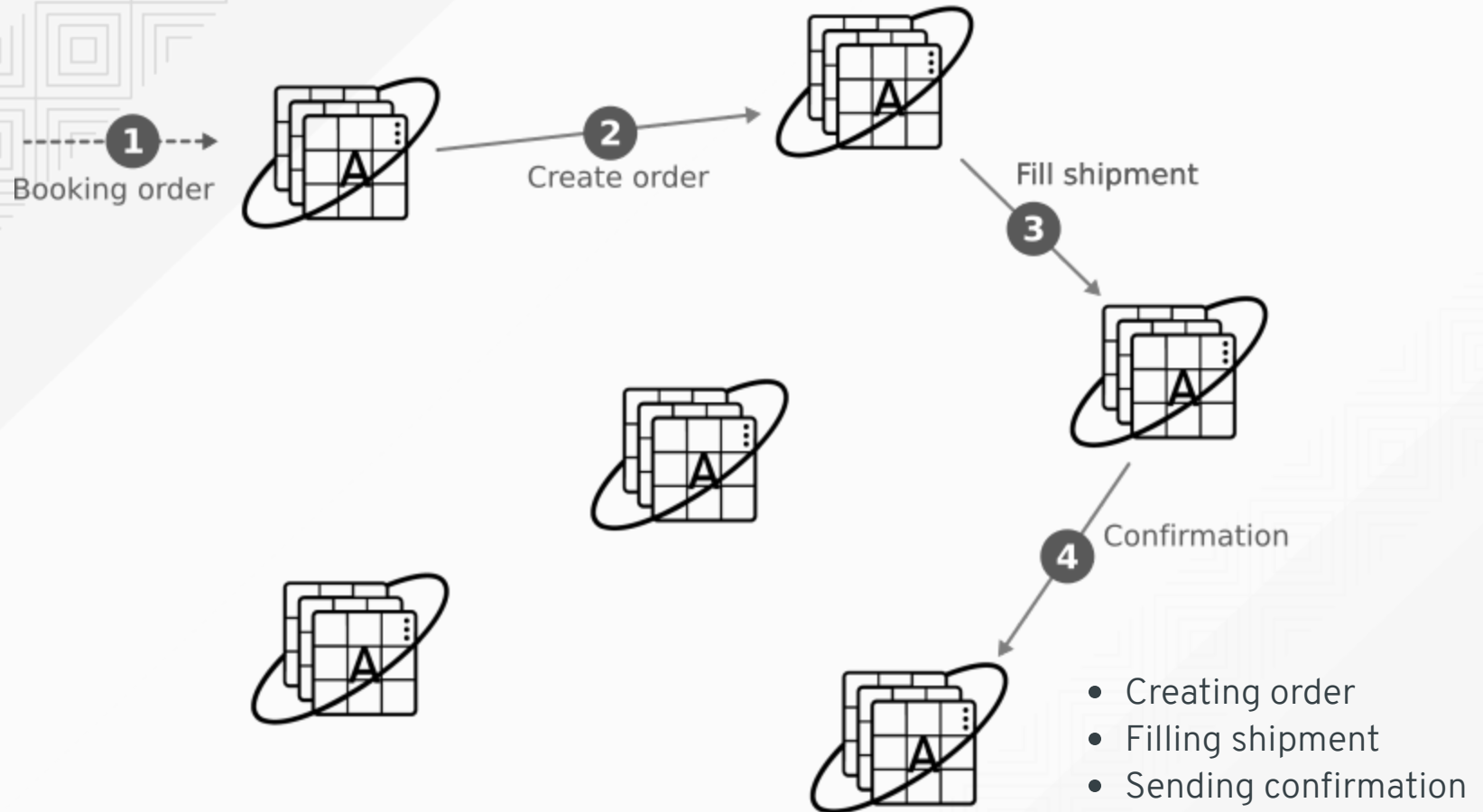


- Creating order
- Filling shipment
- Sending confirmation

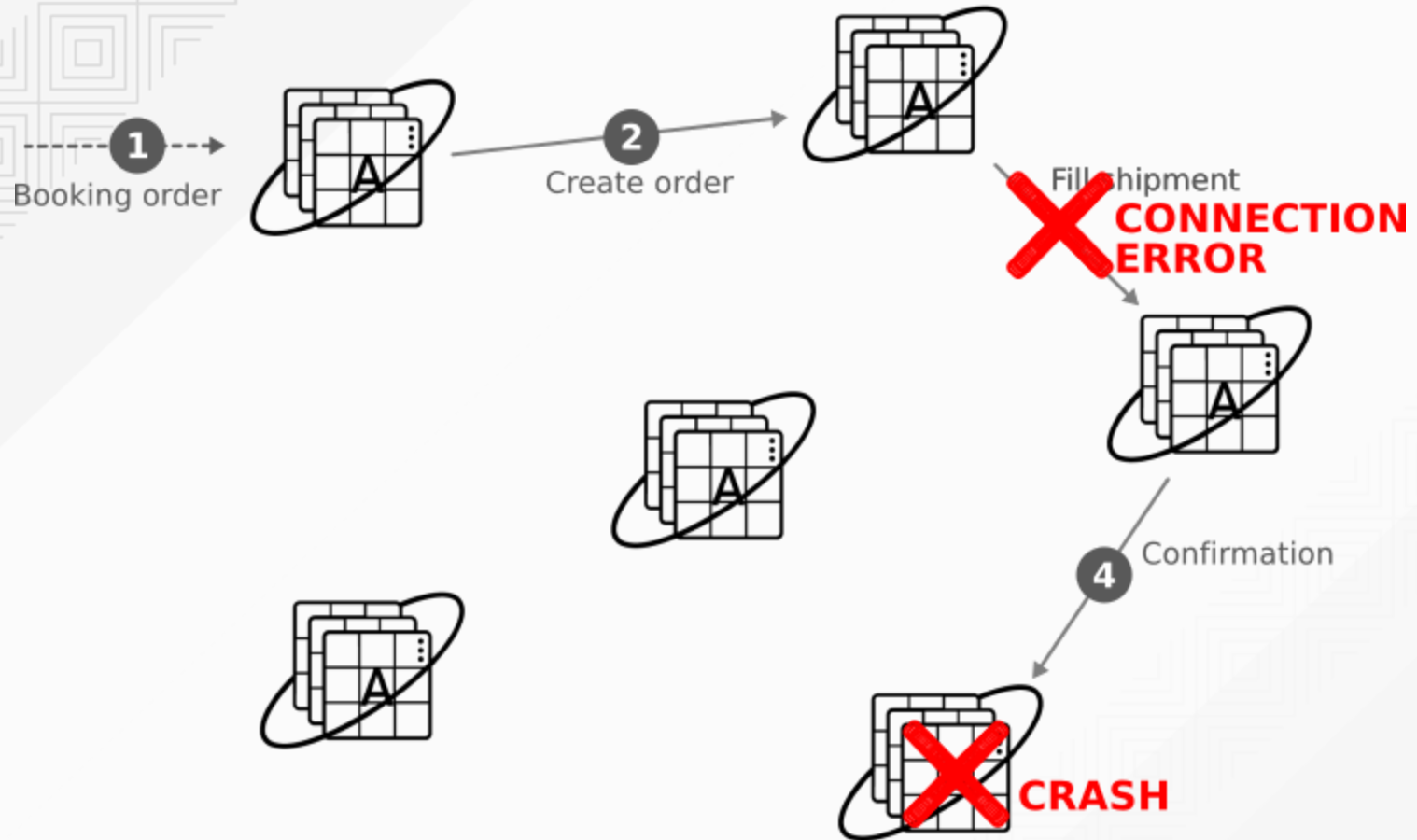
WORLD OF MICROSERVICES



WORLD OF MICROSERVICES



FAILURES HAPPENS



Every sufficiently large deployment of
microservices

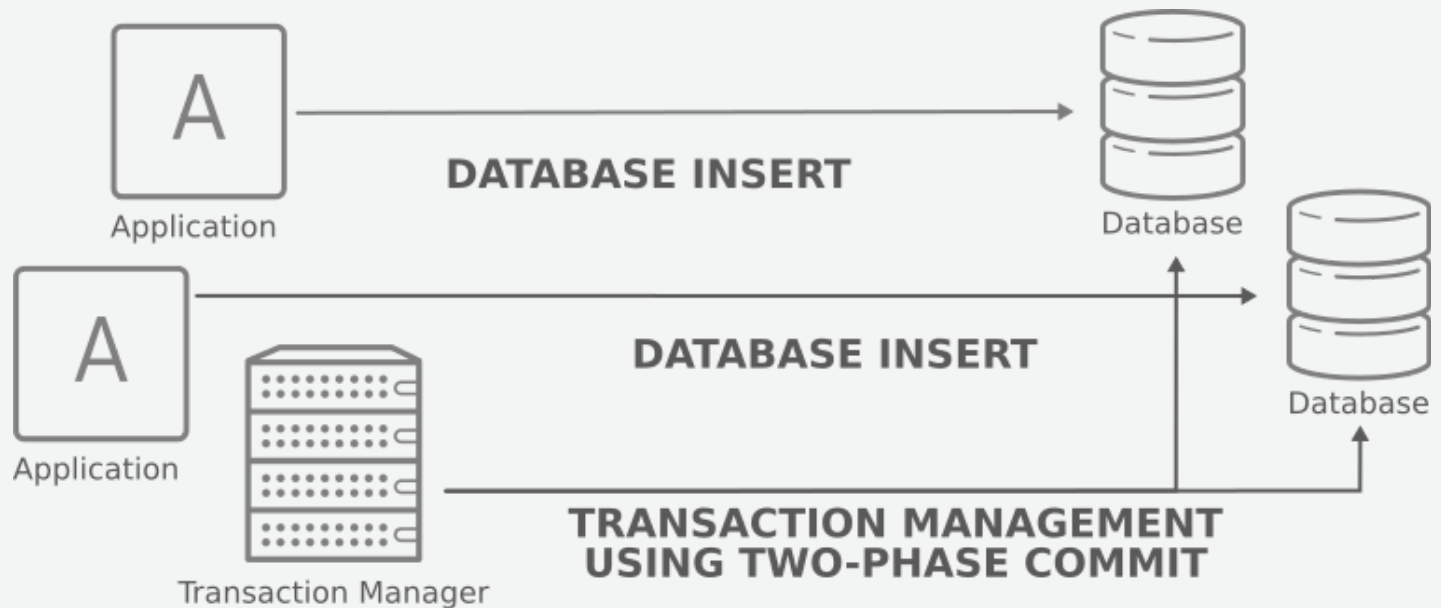
contains an ad-hoc, informally-
specified, bug-ridden, slow
implementation of half of

transactions

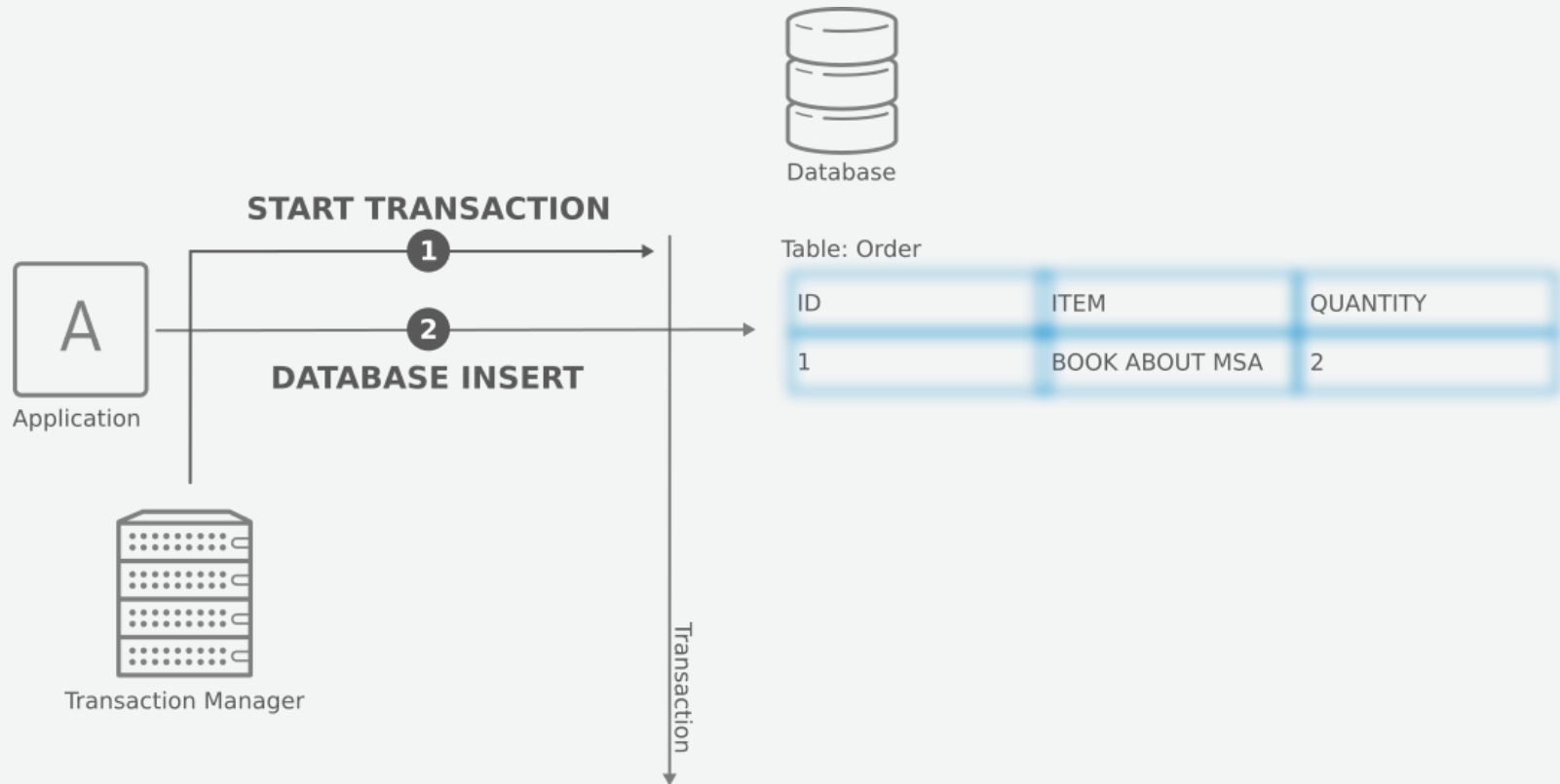
...AND NOW WHAT?

- just use ACID transactions
- but:
 - using locks
 - coupling microservices together

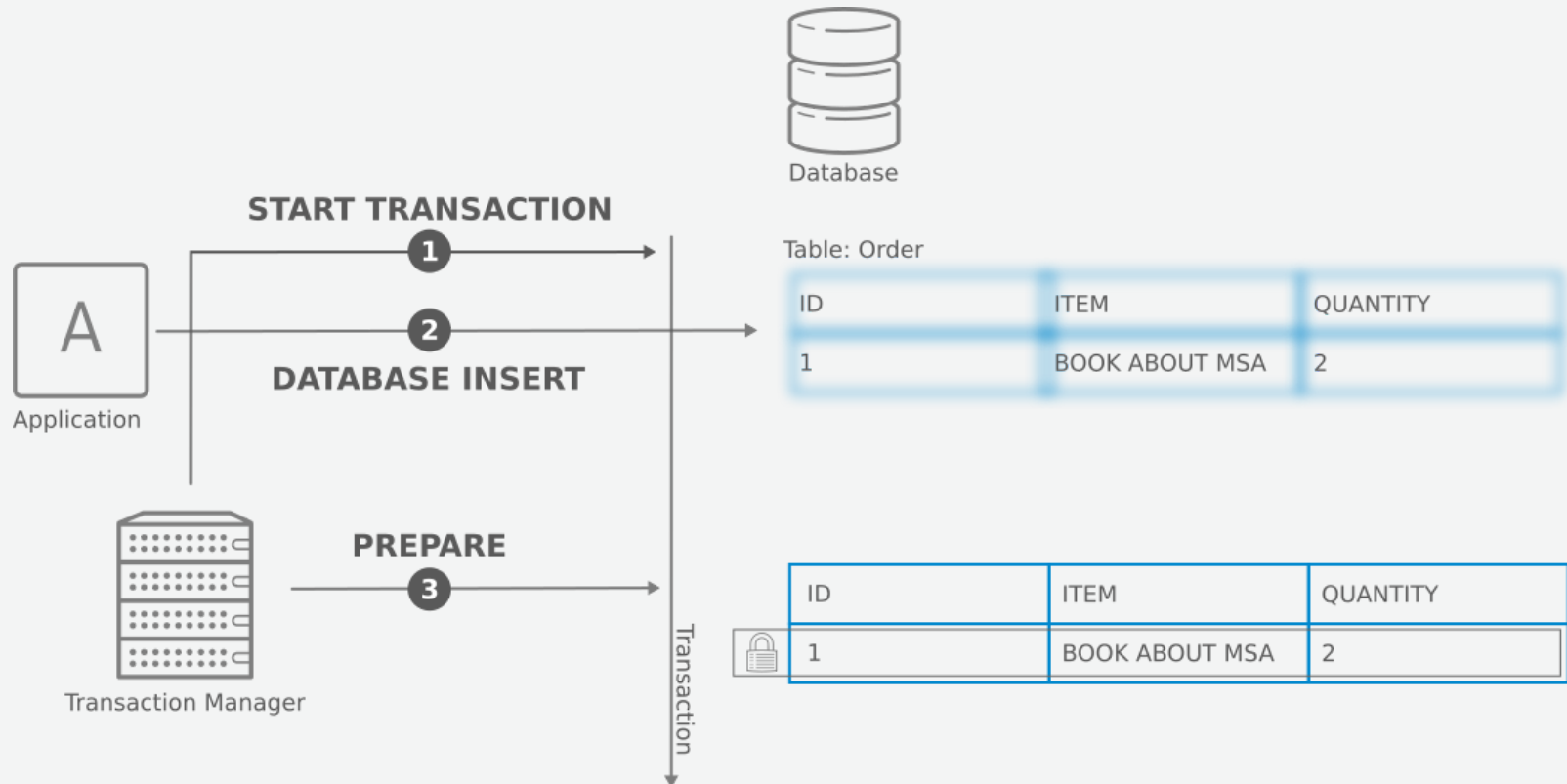
DISTRIBUTED XA TRANSACTION



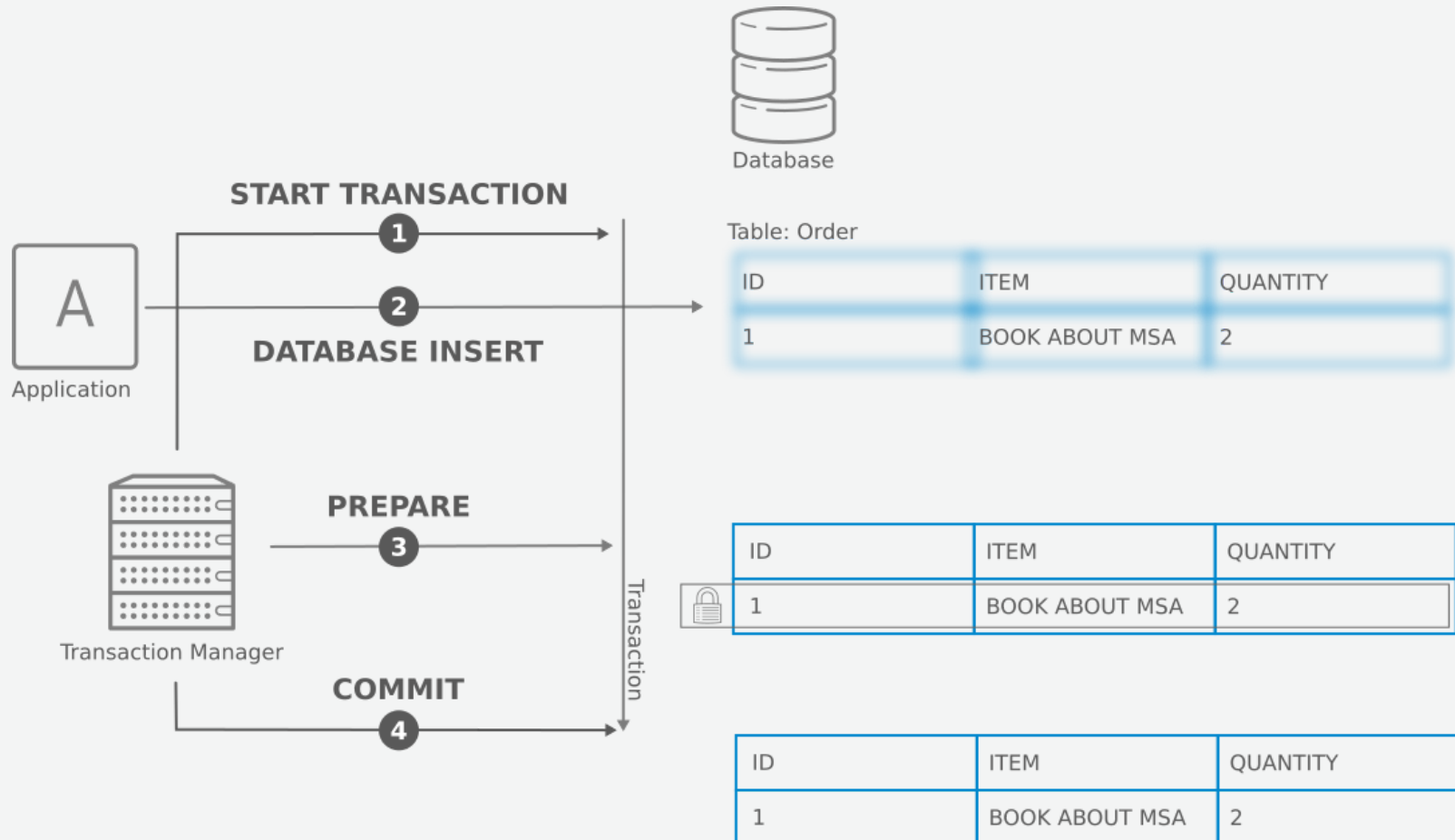
ACID AND TWO-PHASE COMMIT



ACID AND TWO-PHASE COMMIT



ACID AND TWO-PHASE COMMIT



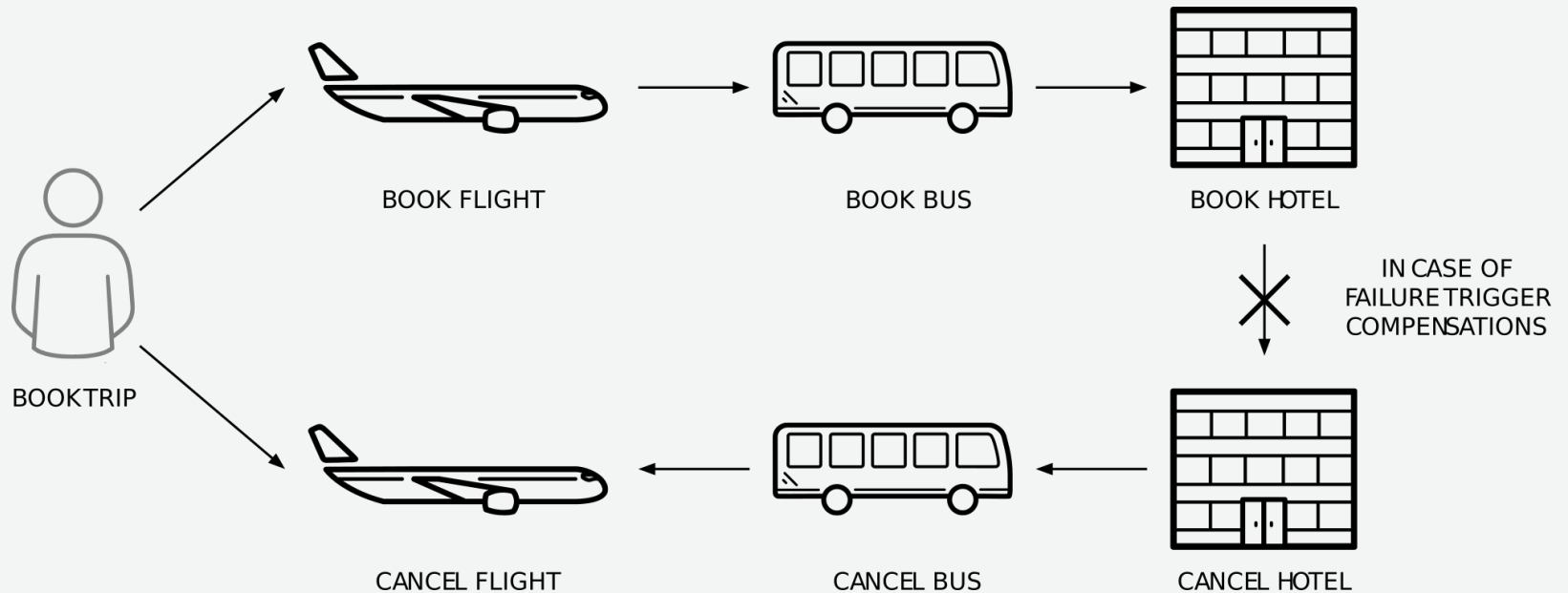
...AND NOW WHAT?

- rollback to monolithic approach
- but:
 - agility
 - independence
 - scalability
 - easy to understand
 - fault isolation

SAGA PATTERN

a distributed domain transaction

SAGA PATTERN



SAGA PATTERN - THE BASIC IDEA

- break overall transaction into **smaller steps**
- steps can be performed in atomic transactions internally
- saga ensures that either the overall transaction is **fully completed** or the changes are undone

SAGA PATTERN

- first published in 1987
- intended for long running transactions in databases
- good fit for microservices nowadays
- two main approaches to saga
 - orchestration
 - provides a good way of controlling the flow
 - an orchestrator tells participants what local transactions to execute
 - choreography
 - each local transaction publishes events that trigger local transaction in other services

LRA: LONG RUNNING ACTIONS

- Java based
- specification proposal for long running activities under Eclipse MicroProfile umbrella
 - <https://github.com/eclipse/microprofile-lra>
- defines LRA coordinator
- over HTTP, LRA context is passed in HTTP headers
- definition for REST style endpoints
- implementation in project Narayana.io

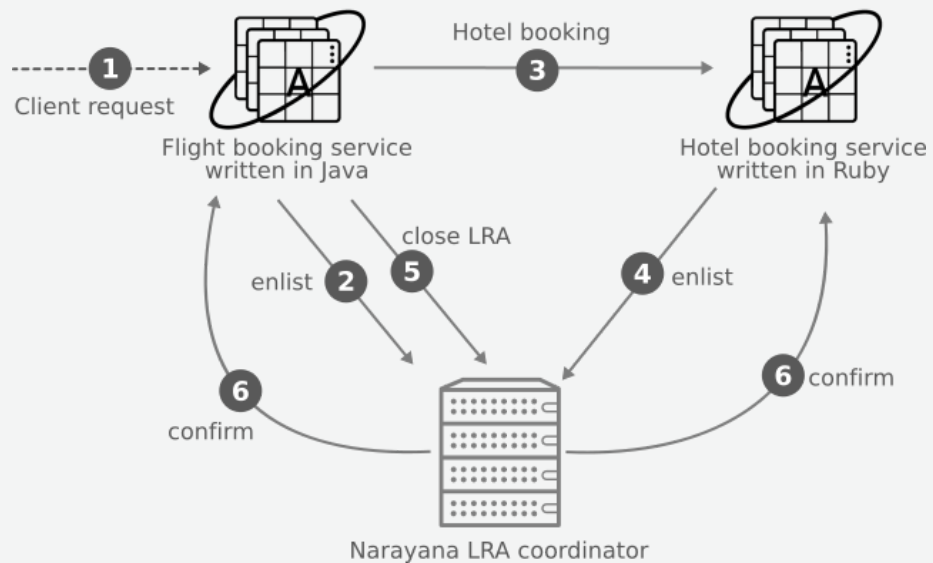


DYNFLOW

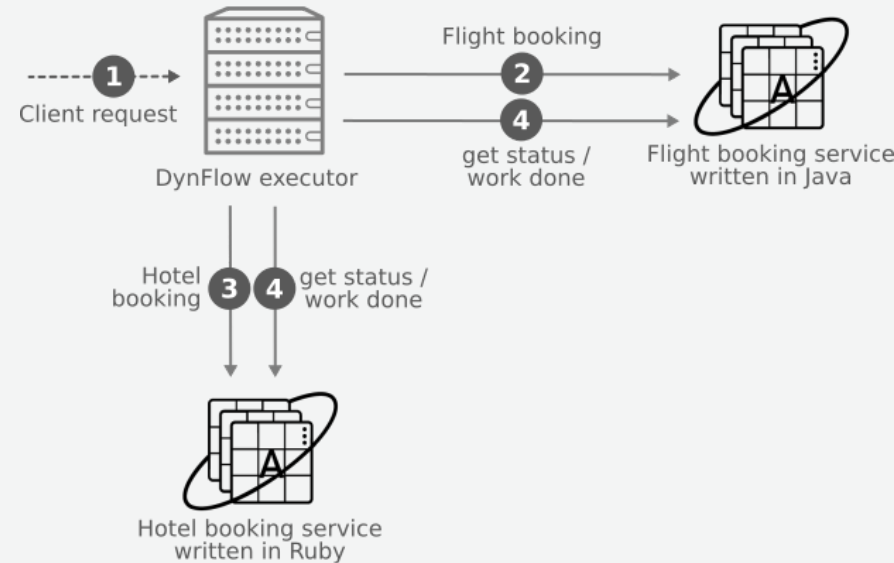
- workflow engine written in Ruby
- currently in use by the Foreman project
- can do all sorts of stuff out of scope of this talk
 - running independent steps concurrently
 - polling external tasks
 - and much more
- support for Sagas in the form of rescue strategy



LRA VS. DYNFLOW

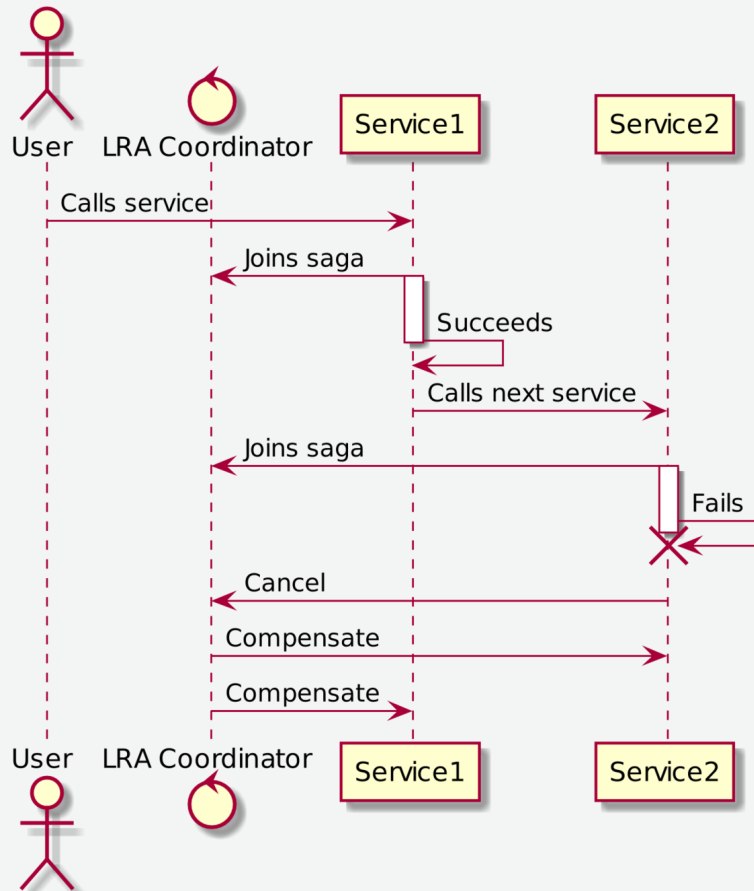


Long Running Actions

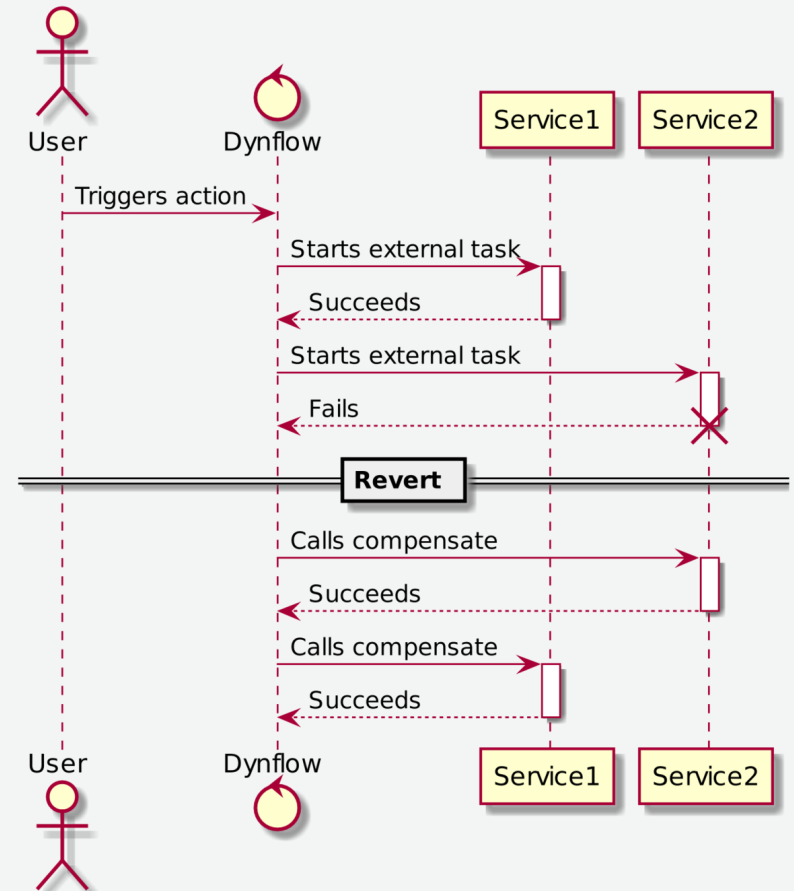


DynFlow

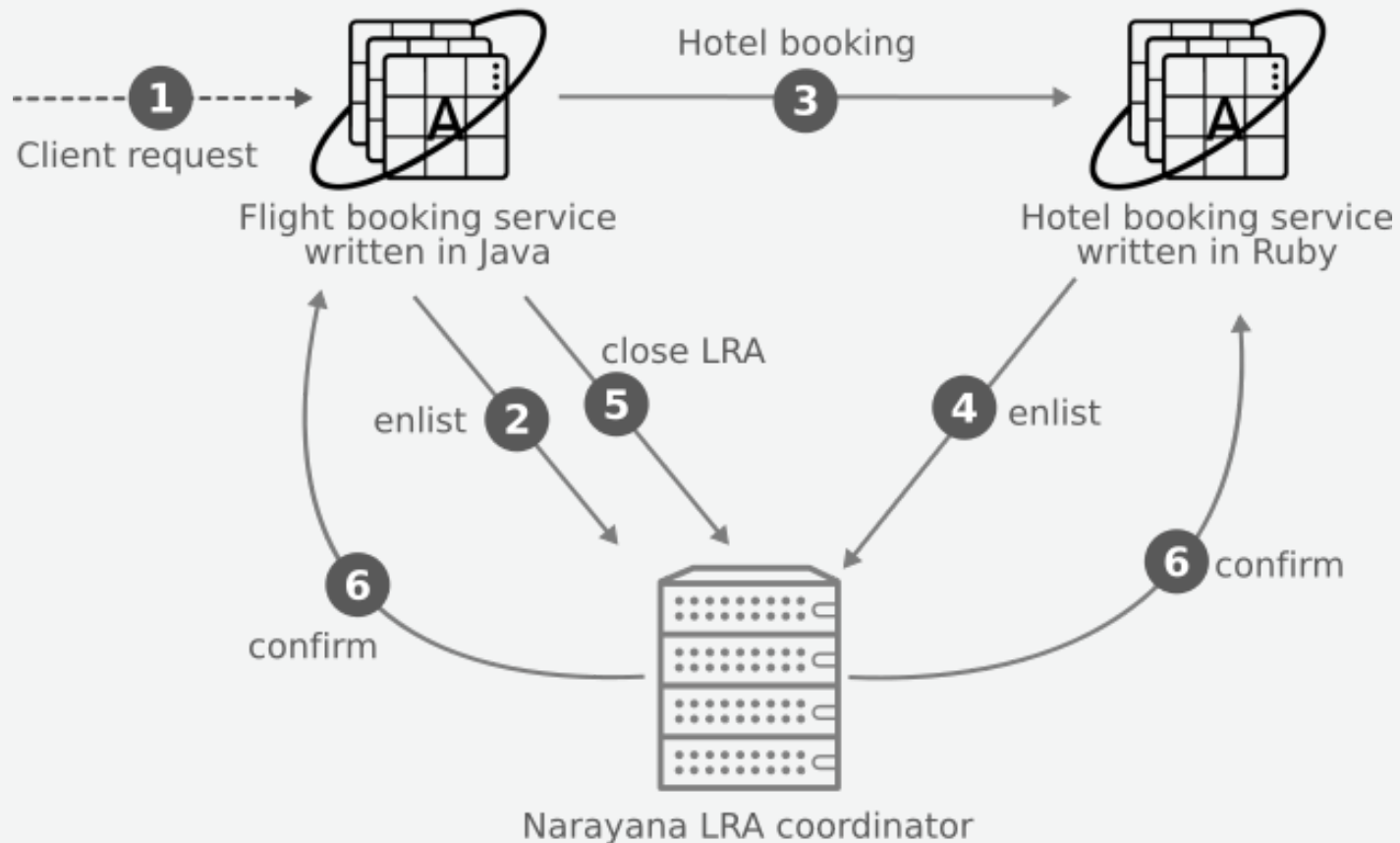
LRA VS DYNFLOW



VS.



LONG RUNNING ACTIONS



LONG RUNNING ACTIONS

```
@LRA
@NestedLRA

@Complete
@Compensate

@Leave
@Status
```

```
org.eclipse.microprofile.lra.client.LRAClient
```

```
startLRA()

closeLRA()
cancelLRA()

leaveLRA()
getStatus()

getAllLRAs()
getActiveLRAs()
getRecoveringLRAs()
```

DYNFLOW BUILDING BLOCKS

- Actions
 - have three phases - plan, run and finalize
 - can be composed
- Execution plans
 - are generated by planning actions
 - in our case a scope for transaction
- Steps
 - units of work

ACTION EXAMPLE

```
class BookHotel < ::Dynflow::Action
  include REST

  def run
    output[:response] = post_rest(input[:url])
  end
end

class BookTrip < ::Dynflow::Action
  def plan
    5.times { plan_action BookHotel, :url => 'http://hotel.california/book' }
  end
end
```

SAGAS IN DYNFLOW

- For an execution plan we know how all its steps finished
- If we know how to undo every single step, we can undo the entire execution plan

ROLLBACKS IN DYNFLOW

```
class BookHotel < ::Dynflow::Action
  include ::Dynflow::Action::Revertible
  include REST

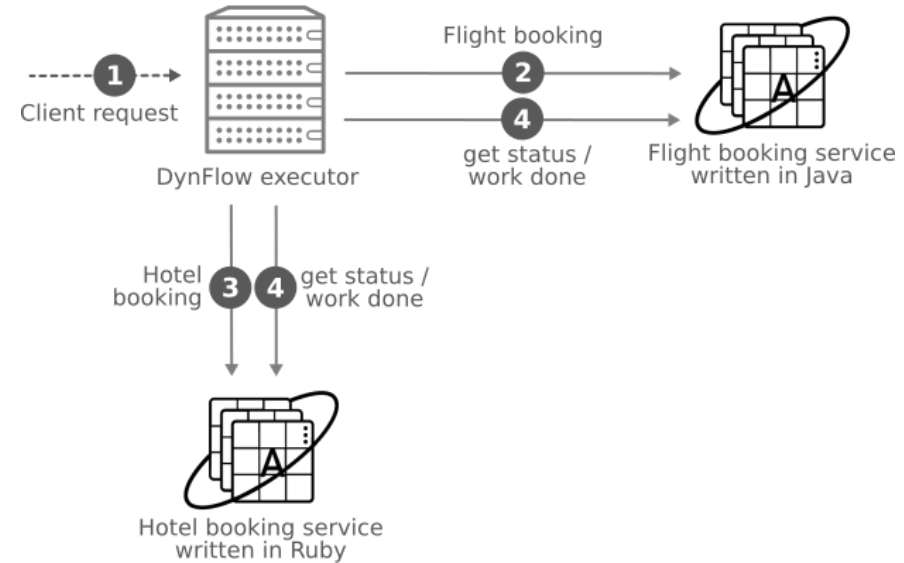
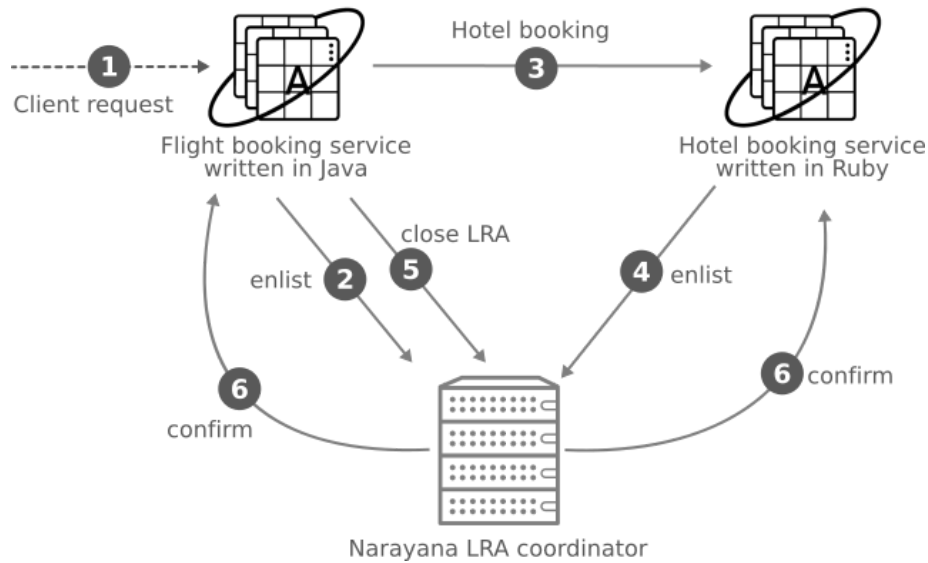
  def run
    output[:response] = post_rest(input[:url], :parse_json => true)
  end

  def revert_run
    id = original_output.fetch(:response, {})[:id]
    post_rest(original_input[:url] + "/#{id}/compensate", :parse_json => false) if id
  end
end

class BookTrip < ::Dynflow::Action
  include ::Dynflow::Action::Revertible

  def plan
    5.times { plan_action BookHotel, :url => 'http://hotel.california/book' }
  end
end
```

DEMO



SUMMARY

- Sagas are great solution for transactions in microservice deployments
 - if you're willing to loosen your requirements and go from strict atomicity to eventual consistency

QUESTIONS

LINKS

- MicroProfile LRA specification: <https://github.com/eclipse/microprofile-lra>
- Community gitter: <https://gitter.im/eclipse/microprofile-lra>
- Blog posts: [Narayana LRA: implementation of saga transactions](#), [Saga implementations comparison](#)
- Link to LRA demo: <https://github.com/ochaloup/devconf2019-lra>
- Dynflow: <https://github.com/dynflow/dynflow>
- Dynflow documentation: <https://dynflow.github.io>
- Saga paper: <https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf>



redhat®

THANK YOU!