3. Assignment 3

In this assignment you will learn how to use message-oriented middleware, container-based virtualization, and stream processing systems to solve problems related to scalability and deriving higher-level knowledge from system events.

Application Scenario

The application scenario for this assignment revolves around the real-time matching of drivers and riders. When a rider requests a trip, a rider should be matched to a driver within a specific region. This matching process is quite resource intensive, as there may be many drivers that we need to calculate a matching score against. Also, we should ensure low wait time for riders even during times when there are many trip requests. It is clear that this part of the system is faced with serious scalability challenges in the face of a growing user base and varying workloads at different weekdays and times or during events. Furthermore, other system components want to be alerted about problems in the process.

Fig. 3.1 illustrates the overall architecture of the system.

Fig. 3.1 Scalable trip matching & alerting system

Our system is partitioned into operational regions. Requesting a trip is represented as an `TripRequest`, a simple message object that contains the region and the trip's pickup geo coordinates. For each region, there can be zero or more workers that handle trip requests for this region. Note that this is a classic producer/consumer problem, where synchronization is commonly solved via work queues. For each worker type there is one queue that holds the trip requests of the respective region. The request gateway receives the trip requests and routes them to the correct work queue. The workers take items from the queue, process them, and then report back to the system by publishing data to a topic associated with the region. A monitoring component monitors the amount of requests in a queue and also uses the published data by the workers to calculate the average time it took to process requests. This data is used by the elasticity controller that dynamically spawns or removes workers based on current demand. The components emit system events that are fed into the stream processing platform that creates alerts when it detects problems in the pipeline.

Note that we will not to actually integrate the system end-to-end (although if you implement all components, it's possible), but instead we will focus on the technologies to implement the individual components.

Points

Task	Points
Messaging	7.5
Container Worker	4.25
Autoscaling Containers	5.25
Event Stream Processing	7.5

3.1. Messaging

Each region is handled by a specific type of worker, as regional laws may require slight differences in how the matching is done. For each region there exists one work queue and multiple matching workers that continuously read from that queue. A request gateway accepts and routes a `TripRequest` to the correct work queue based on the region. The workers then send data back via the publish/subscribe pattern. The system is currently deployed in three regions, which are also encoded in the `Region` enum:

- AT_VIENNA
- AT_LINZ
- DE_BERLIN

As messaging middleware, we will use RabbitMQ that implements the AMQP protocol. Before you get started, we suggest you familiarize yourself with RabbitMQ concepts. The RabbitMQ tutorial website provides excellent guides on how to implement different messaging patterns.

Throughout this task you should use the RabbitMQ Java API. Connect to the RabbitMQ instances running locally `127.0.0.1` using the credentials `dst/dst`. We use the default port configuration of RabbitMQ. The `Constants` class also holds these values. You can access the RabbitMQ admin GUI via a web browser http://127.0.0.1:15672 with the same credentials. From there you can also

send messages to topics or queues and create queue bindings to test your application.

3.1.1. Creating Queues and Exchanges

To create all necessary queues and exchanges required by the system, implement the `IQueueManager` and put your implementation in the `impl` package. Make sure to use the values defined in the `Constants` class when necessary. The `QueueManager` is used in the test cases to create and tear down your queues and exchanges. The easiest way to implement it is to run your `QueueManager` code and verify the effects in the RabbitMQ admin interface.

You may need to come back and update this part several times as you implement the rest of your solution.

3.1.2. Routing Requests

Implement the `IRequestGateway` to route requests to the correct work queue based on the region. RabbitMQ provides a great tutorial on work queues. You can either route your message manually by selecting the queue in your Java code, or you can use a custom exchange and bind the queues to that exchange with the respective request region as routing key.

Because our workers aren't written in Java, to send messages, we first need a platform-independent message format. Serialize the `TripRequest` objects into JSON format using the Jackson library and use the JSON strings as payload.

You can test this part by manually calling your `submitRequest` method implementation, and then navigating to the respective queue in the RabbitMQ admin interface and clicking the "Get Message(s)" button.

3.1.3. Monitoring

In a later task you will implement the elasticity controller that spawns new workers based on the current workload. To facilitate this mechanism, we first need a component to monitor the workload. The workload is determined by the number of requests waiting in the queue, the number of workers that are processing requests, and the average request processing time for each request region.

For this task, you should implement the first two methods of the `IWorkloadMonitor`, i.e., `getRequestCount` and `getWorkerCount`. Calculating the average processing time is part of the next task. Use the RabbitMQ HTTP Client to access the HTTP API via the URL defined in the `Constants.RMQ_API_URL` constant. The API provides several ways of accessing information about queues. You can find details about the API in the admin interface on http://127.0.0.1:15672/api/.

3.1.4. Publish/Subscribe

The monitoring component also calculates the average request processing time by aggregating the data published by workers. When a worker is done processing a request, it publishes a message containing the request id, the result (the id of the matched driver), and the time it took to process the request in milliseconds. A worker publishes messages as JSON into the topic that corresponds to the region of the worker (for this task, our tests emulate the workers and publish messages into the topics). Your monitoring component should subscribe to those topics and calculate for each worker type the average processing time of the last 10 requests.

Read the RabbitMQ tutorial on how to implement the publish/subscribe pattern. First, make sure your QueueManager creates the required topic exchange. The expected name is defined in the `Constants.TOPIC_EXCHANGE` value. As routing key for binding queues to the exchange, use the request type in lower case in the namespace `requests` (i.e., `requests.at_vienna`, `requests.at_linz`, or `requests.de_berlin`). The message returned by the workers is formatted as follows:

```
{"requestId":"abcd-1234-...","driverId":"1234","processingTime":"4204"}
```

You should deserialize the messages into the `WorkerResponse` DTO using Jackson.

Make sure that your implementation fulfills the following requirements (think about how you need to bind queues to the exchange):

- consider that the topic exchange may be used for other routing keys unknown to your component. You may, however, assume that the `requests.` routing key namespace is used only by the workers to publish the request processing data.
- any queues you bind to the exchange for each instance should be cleaned up properly after the instance is closed

- it should be possible to run multiple instances of your monitoring component in parallel

3.2. Autoscaling Containers

Container-based virtualization is an important technology to solve problems of modern distributed systems application. Being able to isolate applications and package them in a platform-independent way has great benefits (also drawbacks, but we will focus on the benefits for now)! In particular, container platforms like Docker and Docker Swarm greatly simplify autoscaling solutions, where the system reacts to changing workload to maintain a certain quality of service or responsiveness. In this task, you will solve a simple autoscaling problem for the workers of our system using Python, Docker, the Java Docker API, and RabbitMQ.

3.2.1. Python Worker

First, you will implement the matching worker as a containerized Python 3 application. Familiarize yourself with Docker and how to 'dockerize' an application using a Dockerfile and the `docker build` command. This article gives an excellent overview on how to do this for Python applications in particular.

A worker processes trip requests (continuously until stopped, one at a time) and is associated with a specific region that the worker is responsible for. Implement the `worker.py` file in the `ass3-worker` module. It should take one string argument which is the region, e.g., `worker.py at_vienna`. Multiple workers for the same region should be able to run in parallel containers.

A worker performs the following tasks:

- Log what is happening into stdout (via `print`)
- Upon starting the script, print the region.
- Connect to the RabbitMQ server
- If the connection to the server fails, the worker should wait for 5 seconds and then try again. This loop should run indefinitely until a connection is made.
- Take an `TripRequest` from the correct RabbitMQ queue (according to the type)
- Record the processing start time before starting

- Query all available drivers and their geo locations from Redis (the tests emulate the component that manages these data)
- Do region specific matching (in our case, we will only match the geo distance between the driver and the pickup location)
- Remove and report the driver that is closest to the rider
- Calculate and report the processing time (in milliseconds) and the result into the correct RabbitMQ topic
- Repeat taking elements from the queue until the container is terminated

To write a RabbitMQ application in Python, make use of the pika Python client. Connect to the RabbitMQ host using the same credentials as in the previous task.

We take a straight forward approach to matching drivers and riders. The worker finds from all available drivers in the region, the driver closest to the pickup location. (This is basically how Uber did matching for many years until they introduced what they call 'batched matching', which you can read about on their marketplace site.). A different service (which our tests will emulate) manages the currently available drivers and their current coordinates in Redis. Available drivers and their coordinates are stored in Redis hashes `drivers:region` where `region` corresponds to the specific region (e.g., `drivers:at_vienna`). The key holds a hash that maps the driver id (e.g. `32314`) to their current GPS coordinates in the format `latitude longitude` (e.g., `48.19819 16.37127`). To access Redis, use the redis-py library. The code template contains a file `ass3-worker/redis-data.sh` which provides some driver test data and can be executed locally. The worker should fetch all available drivers, and calculate the geographic distance between the driver's location coordinates and the pickup coordinates. To do that, you can either use an existing Python library, or calculate it yourself using the Haversine formula. Before creating the response, remove the driver from the Redis hash. If the driver no longer exists (e.g., because it was removed by a different worker), re-run the matching process for the request and reset the processing time. Hint: you can use the return value of HDEL. In case there are currently no available drivers in the hash, return an empty string as driverId, and 0 as processing time.

In our scenario, workers would probably perform more complex region-specific matching calculations, meaning that each type of worker would have its own Python application. To simplify things, you just need to have one Python script that takes the region that it handles as an input parameter. To 'simulate' an additional workload, use time.sleep(secs). For each request, select a random integer in the following ranges for the respective region: at_linz (1-2), at_vienna (3-5), de_berlin (8-11), and call the

sleep method. Note that these values are simply selected in a way that makes them useful for testing. The processing time should include the time it took to fetch all drivers, do the matching, and the additional simulated workload.

Later, when we stop the Docker container running the application, Docker sends a `SIGTERM` signal to the application. To stop a worker gracefully, use the `signal` module for signal handling and attach a handler for the `SIGTERM` signal. Again, in a real implementation, we would do proper cleanup of the worker and probably return the request into the queue.

Warning

For testing purposes, you must print `"Signal received!"` when the signal handler was called, and then exit the application using `sys.exit(0)`.

3.2.2. Dockerfile

For this task you should write a `Dockerfile` (in the `ass3-worker`) for the Python worker. We use the `python:3-slim` base image to create our worker Docker *image*. That image will then be used to spawn instances of our worker as Docker *containers*. Because the containerized application requires Python and the pika library (and any other python libraries you may need), but the image does not contain these libraries, you need to install it when creating the image. This can be done by running `pip install` (e.g., `pip install pika`) when the image is created (define it in the `Dockerfile`, you will need an Internet connection to build the image because it has to download dependencies from the Python repos).

Once you have your Dockerfile, you can build your Docker image. With the setup we have, there are various ways to do this.

You can directly build your dockerfile from your repository.

When you build your image, make sure to tag it as `dst/ass3-worker`. You can do this by adding `-t dst/ass3-worker` to your build command. We should be able to spawn a container with your worker application using that container name.

Once you have your image, you now run and test it manually using the `docker run dst/ass3-worker` command. Add the `--rm` flag to remove containers immediately after they exit, otherwise they remain on the Docker host (see via `docker ps -a`) You can pass arguments to the program running inside the container

by simply adding the arguments to the end of the run command. For example, `sudo docker run --rm dst/ass3-worker at_vienna` should start a worker for the region 'at_vienna'.

Note

Make sure that your containers are running in the same network. The services defined in the docker-compose file are all running under the network "dst". With that you can make use of the hostnames "redis" and "rabbit" to connect your container with the previously named services.

3.2.3. Container Management

Now that we have our images, we will implement a service to control containers from our Java application. Implement the `IContainerService` component and make use of the Java Docker API. You can find code examples in the docker-java Getting Started docs or in this article. The `dst/ass3-worker` Docker image should be available on the host. When we test your solution we first build your Docker image as described previously.

Your container service should be able to

- return the metadata of containers currently running (by the worker type)
- spawn a new container for a given worker type
- stop a specific container
- remove the stopped container

Make sure that your containers are removed after they are stopped. Stop a container using the stop command, not the kill command! As described earlier, when a container is stopped via the stop command, the application receives a `SIGTERM` signal and is expected to gracefully terminate.

Note

As before, make sure that your solution deploys the containers into the "dst" network. With that you can make use of the hostnames "redis" and "rabbit" to connect your container with the previously named services.

3.2.4. Elasticity Controller

Now, implement the `IElasticityController` to automatically scale the number of workers running in the system. The elasticity controller makes use of (a) the

monitoring data provided by the `IWorkloadMonitor`, i.e., the amount of waiting requests, the average request processing time and the amount of current workers, and (b) the `IContainerService` to scale-out and scale-down containers.

A call to `adjustWorkers` should scale the amount of workers to make sure that a request does not wait longer than a specific amount of time. When exactly the system should call this method is a completely separate topic that we will leave out to contain complexity. You can assume that the method is called only after a scaling cooldown period.

For each region, we define the following values that we can get directly from our monitoring system and a specification:

- k: number of parallel workers currently running
- q: number of waiting (queued) requests
- r^10: the average processing time of the last 10 requests
- rmax: defined maximum wait time for a request

We specify rmax, i.e., the maximum time a request is allowed to be queued, for each region as, at_linz: 30 seconds, at_vienna: 30 seconds, de_berlin: 120 seconds (in a real application these values could come from estimations or historic data).

The general goal of the autoscaling is to make sure that the expected wait time of the last request in the queue averages around rmax, illustrated in Fig. 3.2, by adjusting k accordingly. To facilitate this, we further define the following values:

- rexp: expected wait time for the last request in the queue (calculate by using k, q, and r^10)
- α: scale-out threshold (specified as 0.1)
- ω: scale-down threshold (specified as 0.05)

Fig. 3.2 Processing time over time and scaling threshold.

The system should tolerate a slight variations in rexp, and should therefore only adjust k if the difference between rexp and rmax exceeds the given thresholds. That is, only scale-out when $rexp > rmax*(1+\alpha)$, or scale-down when $rexp < rmax*(1-\omega)$.

For the tests we inject mocked instances of `IContainerService` and `IWorkloadMonitor` that you should call. We then check

whether your calls to `IContainerService` (spawn and stop) are correct. Feel free to also add your own tests where you inject your implementations to test the integration of your solution.

3.3. Event Stream Processing

In this task you will implement an alerting system using the stream processing platform Apache Flink.

Note

The project is configured to use Flink v1.14, please be aware of this when looking through the docs and other blogs, etc. Flink's API can change drastically between releases. All links below are pointing towards the 1.14 version on purpose.

Stream processing is a computing paradigm that has gained immense popularity in the past decade. In this paradigm, a stream of data is filtered, transformed, or aggregated using stream operators in real time. Complex event processing (CEP) is a related concept that is used to derive higher-level knowledge from events as they occur. Systems like Apache Storm, Apache Heron, or Apache Flink, are widely used in both research and industry. You will use Apache Flink's stream and complex event processing facilities to detect problems within the trip pipeline based on the life-cycle of a trip request.

We assume that our system emits event data (`ITripEventInfo`) when the state of a trip changes (this could also easily be implemented by, e.g., using Rabbit MQ topics). Based on these incoming `ITripEventInfo` data you will:

- calculate the duration of a trip matching
- calculate the average trip matching duration in a region
- emit warnings about timed-out trips
- emit warnings about failed trip matchings, and
- emit alerts if a certain number of warnings have been received for a region.

The classes/interfaces you need to implement are: `IEventProcessingEnvironment` which creates the stream processing graph; `IEventSourceFunction`, the data source function that feeds `ITripEventInfo` objects into the stream; and `EventProcessingFactory` which instantiates your implementations.

Note that `IEventProcessingEnvironment` has several setters for `SinkFunction`. You should add these sinks to the respective data streams you will be creating in subsequent tasks. They are set by the tests to verify your solution, and are all set before `initialize` is called.

Please be aware that there are some limitations when using Java Lambda Expressions in the Flink API. Queries sometimes don't work or fail with some obscure exceptions. For more information on this topic you can read the respective chapter Java Lambda Expressions in the Flink documentation.

3.3.1. Event Source

We assume that the underlying system emits `ITripEventInfo` data. To feed the Flink data stream, first, write a source function that receives `ITripEventInfo` objects from an `EventSubscriber`. To that end, implement the `IEventSourceFunction` interface. Connect to the `EventPublisher` using `EventSubscriber.subscribe(SocketAddress)` using the port defined in the `Constants` class. Note that all Flink operators have to be serializable, so make sure your source function is also. Use the open and close methods to initialize and close any resource you need. Have a look at `RichSourceFunction` implementations of Flink, and keep it simple!

3.3.2. Simple Transformation & Watermarking

Using the previously created source as input, all `ITripEventInfo` objects that do not have a region assigned to them (where `region` is null) should be filtered. The remaining objects should be translated into `LifecycleEvent` objects.
A `LifecycleEvent` indicates a change in the state of a streaming event. The `region` of a `LifecycleEvent` refers to the region that the trip was requested in. The result should be a `DataStream` with the generic type `LifecycleEvent`.

Real-time stream processing systems with time-based window semantics (discussed later) have to deal with events that arrive late or out-of-order. Flink supports different notions of time. We assume that our system guarantees that events arrive in order, however, you need to make Flink aware of the real timestamp, the one in the `ITripEventInfo` object, by adding a `WatermarkStrategy` to the `LifecycleEvent` data stream. This consists of a `TimeStampAssigner` and a `WatermarkGenerator`. Implement a punctuated `WatermarkGenerator` and a `TimeStampAssigner` that that extracts from a given `LifecycleEvent` its timestamp (which was passed from the `ITripEventInfo`).
The documentation page offers detailed information on the process of generating

Watermarks. Make use of convenience methods provided by WatermarkStrategy, to supply a `WatermarkGenerator` and assign timestamps.

Note

During the discussion sessions, you should be able to explain the general idea behind *event time* and *watermarks*, and why they are necessary for stream processing platforms.

3.3.3. Complex Event Processing

Use the `LifecycleEvent` stream and Flink's CEP facilities to implement the following two queries:

- **Calculate the duration of the matching process:**
  - When a trip (identified by the `tripId` property) transition from `CREATED` to `MATCHED`, emit a `MatchingDuration` instance. The duration value is the time it took for the event to reach the state `MATCHED`. Note that the lifecycle may include a `QUEUED` transition between `CREATED` and `MATCHED`.)
  - If the event does not reach `MATCHED` within a given time window (specifically, the time specified by `setMatchDurationTimeout`), then issue a `MatchingTimeoutWarning`.
- Detect anomalous trip requests: if a trip request has been matched three times but was always re-queued, that is, transitioned three times between `MATCHED` and `QUEUED`, emit a `TripFailedWarning`.

3.3.4. Stream Window Operators

Now, use stream windows to implement the following two operators:

- Trigger an `Alert` for a region (indicated by the `region` property) as soon as three `Warnings` (of any type) have been received for that region since the last alert.
- Trigger an `AverageMatchingDuration` event for a region, that calculates the average value of the last five `MatchingDuration` instances of that region.

Hint

Due to the discovery of a bug caused by using Enums as key, Flink throws an exception if you use the Region Enum as key. Therefore, use the here described

workaround by using the value of *name()* as key and instantiate the enum with *Region.valueOf(…)*.