# Microservice Inter-communication

# Learning objectives

- Upon successfully completing this work, you should be able to:
  - Explain the differences between synchronous and asynchronous message patterns
  - Discuss, a use-case explaining the benefits and challenges of using synchronous messaging
  - Discuss, a use-case explaining the benefits and challenges of using asynchronous messaging
  - Explain the difference, with a use-case, of provider / consumer, and publisher / subscriber patterns
  - Implement synchronous and asynchronous message patterns for inter-microservice communication

# Inter-microservice communication

Public internet
Azure private network
Docker private network
Virtual machine
Application container
Rabbitmq combined container

80 / 443
kong
8445
public IP:
private IP:

/categories
/question/:category
/moderate
/categories
/categories
/submit
/docs

4000
question
3000
/categories

4100
moderate
3100
/categories
submitted question

4200
submit
3200
categories

4002
MongoDB
27017
mongo vol

4002
MySQL
3306
mysql vol

categories

4102
rmq-admin
15672

4201
SUBMITTED_QUESTIONS
5672

4202
rmq-admin
15672

public IP:
private IP:

4001
etl
3001
moderated questions

4101
MODERATED_QUESTIONS
5672

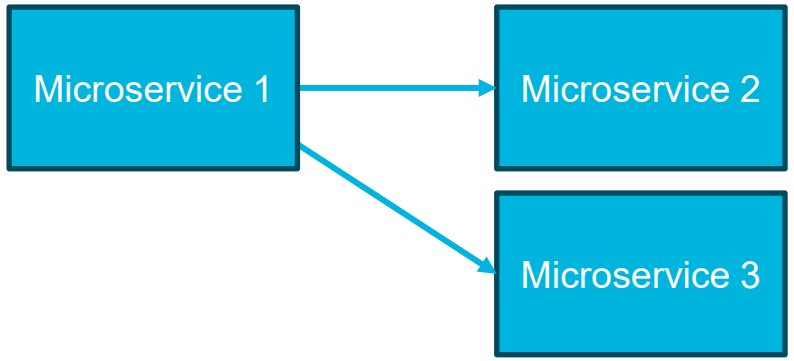
public IP:
private IP:

public IP:
private IP:

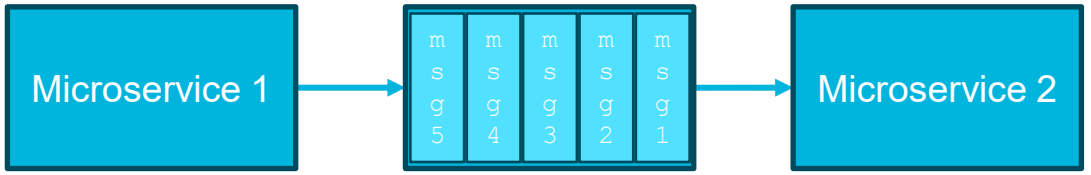
Container port number
Application port number

# Inter Microservice Communication

- HTTP is synchronous
  - JavaScript overcomes thread  blocking with callbacks or promises but an http request still waits for a response, so http is synchronous
  - Synchronous HTTP API is fine in some circumstances but does more tightly couple the services than an asynchronous approach - e.g. if the endpoint changes, it affects all the callers as the contract is broken for all of them or at best, inconsistent . Could build-in versioning into the path to minimise this
  - Response is generally fast as it's a direct point to point call
  - What about destination service failure? Http doesn't account for this unlike SOAP. e.g. you post some data and get a 500 response - what has happened to the data? How do you recover? More code needed
  - What about adding another microservice destination? e.g. You are already calling a "Change of circumstances" microservice with say an address change. A new payment service, also requiring the change of address, would result in a new interface from the caller



- Asynchronous messaging may be a better option in some cases
  - A microservice posts to a queue and that's it. The response is immediate, but the message must wait
  - Ms 1 doesn’t need to know if ms 2 can cope with a burst of messages that it can't cope with
  - The microservices are decoupled - the contract is with the queue. Ms1 need know nothing about Ms2
  - If Ms2 goes down, Ms1 doesn't know or care
  - Shared message queue pattern, as illustrated below, typically uses a dynamic FIFO data structure - a queue



Message broker

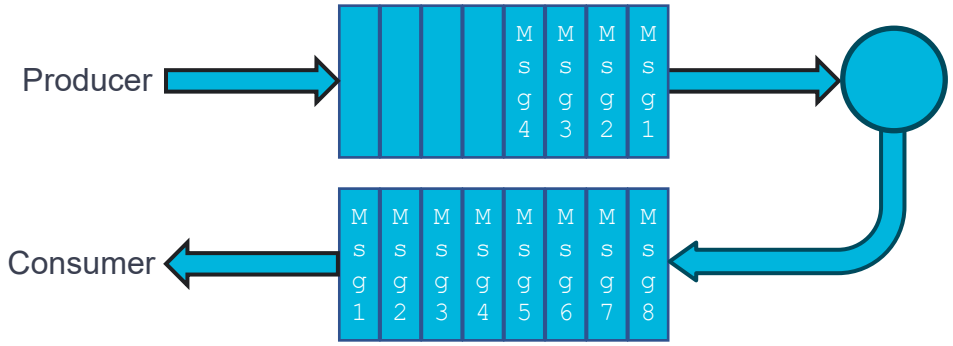
# Message brokers

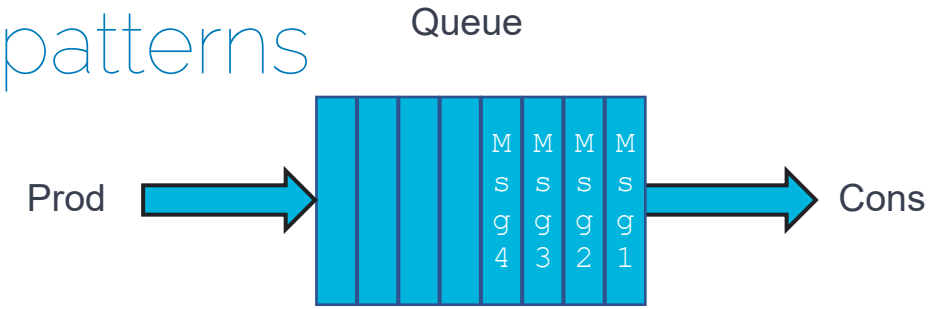
- Message brokers implement the asynchronous message pattern. There's quite a selection:
- RabbitMQ - open source
- Apache Kafka - open source
- Amazon Simple Queue Service (SQS) - AWS
- Azure message broker and Azure storage queue (ASQ)
- and more …

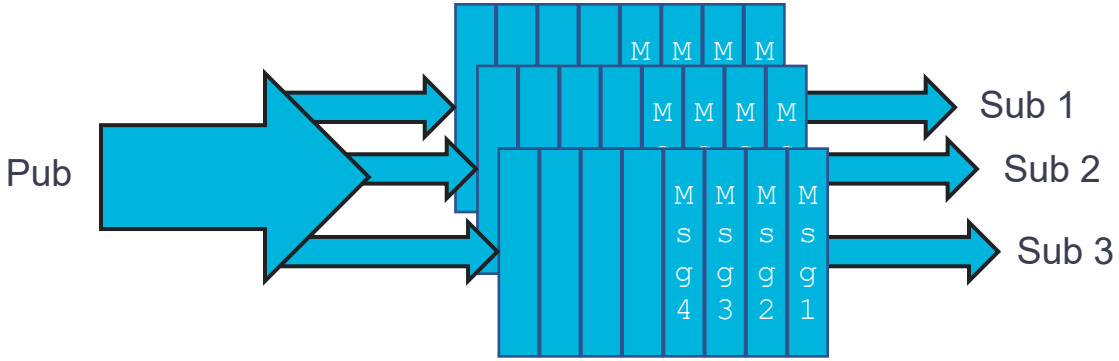
- We will have an introductory look at RabbitMQ and Apache Kafka as they are cloud agnostic
- However, if all-in with a particular cloud provider, their services may be preferential

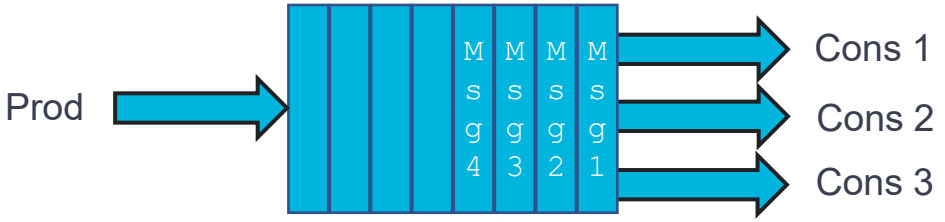
# Producer/Consumer and Pub/Sub patterns

- Asynchronous service to service messaging
- One producer and one consumer
- Use fan-out for pub/sub - adding a new microservice
- Generally, uses push to consumer
- Resilience is built-in to avoid SPOF
- Implemented using a message broker
- Several patterns

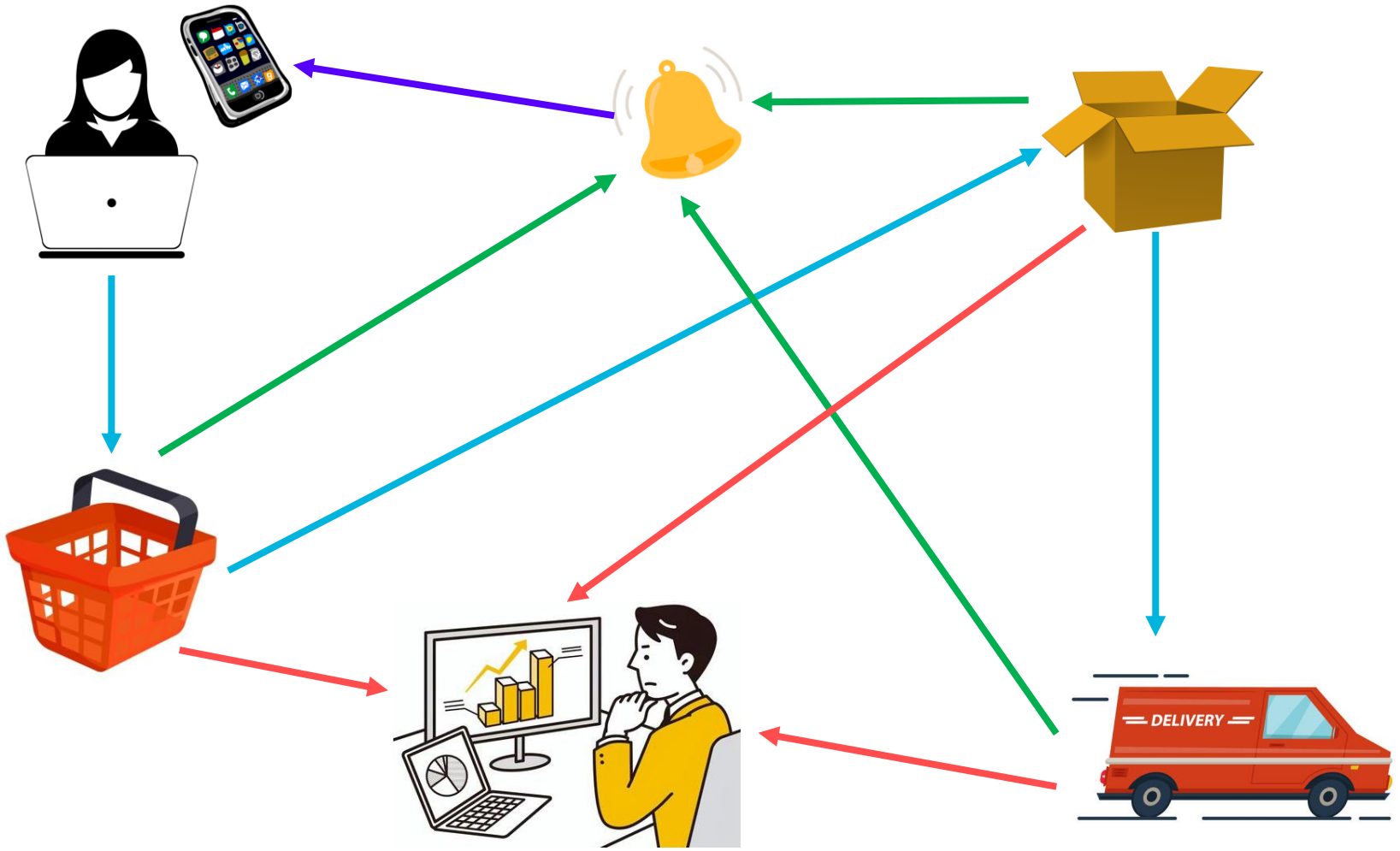


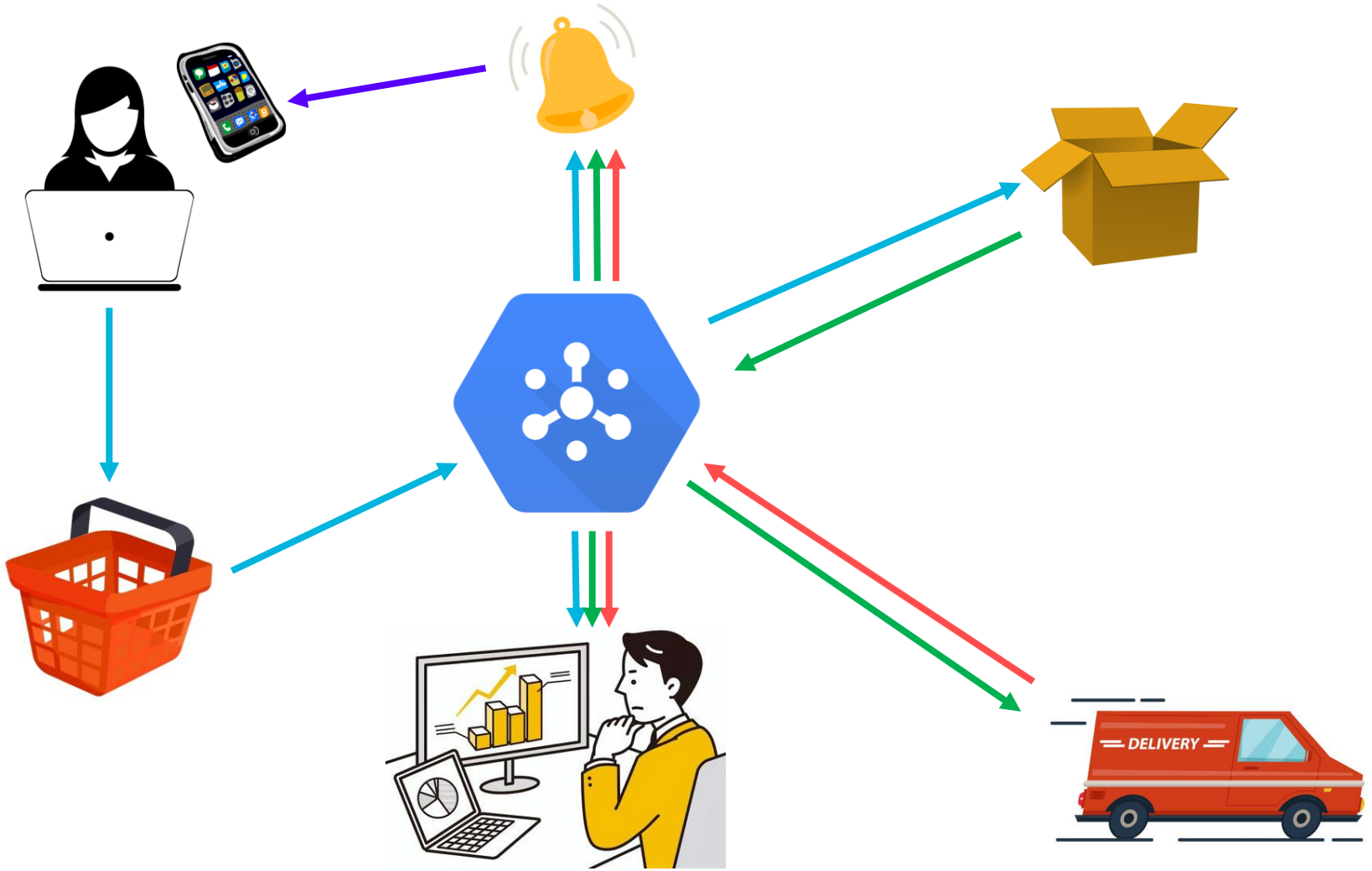



# Simple synchronous scenario



# Publisher / Subscriber asynchronous alternative



# RabbitMQ terminology

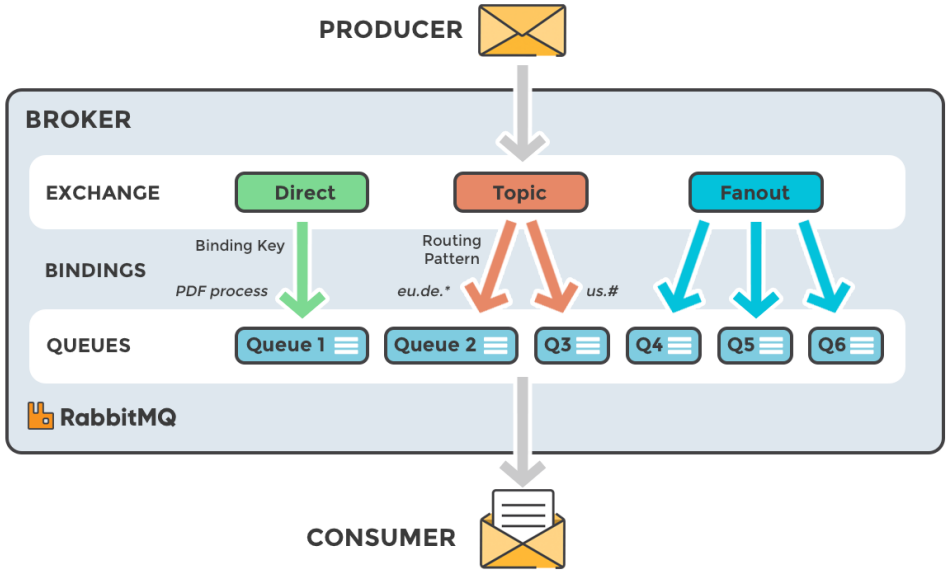
- RabbitMQ implements the Advanced Message Queueing Protocol (AMQP) standard which is a platform agnostic protocol to transfer information between applications. The source and dest are: producer and consumer
- Connection
  - This is a network connection between an app and the queue. Similar to a database connection
- Channel
  - virtual connection inside a connection. Multiple channels can be created within a single connection, allowing for concurrent operations and communication between different parts of a client application
  - For example, you might have one channel for produced messages and another for consuming messages within the same connection to RabbitMQ
- Queue
  - A FIFO-based message storage and delivery component
  - It holds messages sent by producers until they are consumed by consumers
- RabbitMQ is available as a docker container
  - It can implement queues and message routing through a message exchange component which can route messages to various queues based on a routing key
  - It has a management interface to setup and monitor the queues

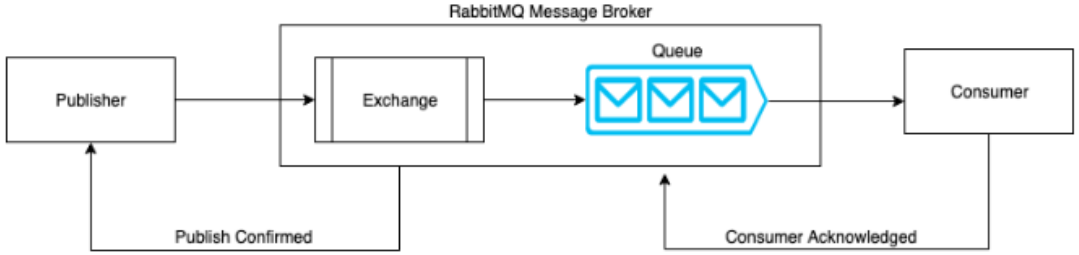
- Producer / Publisher
  - An application that sends a message to the broker. The broker confirms the message is queued
- Consumer / Subscriber
  - Application that consumes the message. Consumer informs rabbitmq that it received the message (acknowledges it) so it can be removed from the queue. Without ack, the message stays where it is
- Exchange
  - Receives messages from producers and routes them to the correct queue based on rules associated with a routing key
- Routing key
  - This can be a word, series of words, wildcard. e.g. all messages that contain pr* as a post code are sent to a queue for consumers of Preston based messages - e.g. for a targeted mail shot
- Virtual host
  - Segregates applications using the same rabbitmq instance. Think of it as separate broker instances each with different permissions and credentials but built into a single instance, e.g. a single container

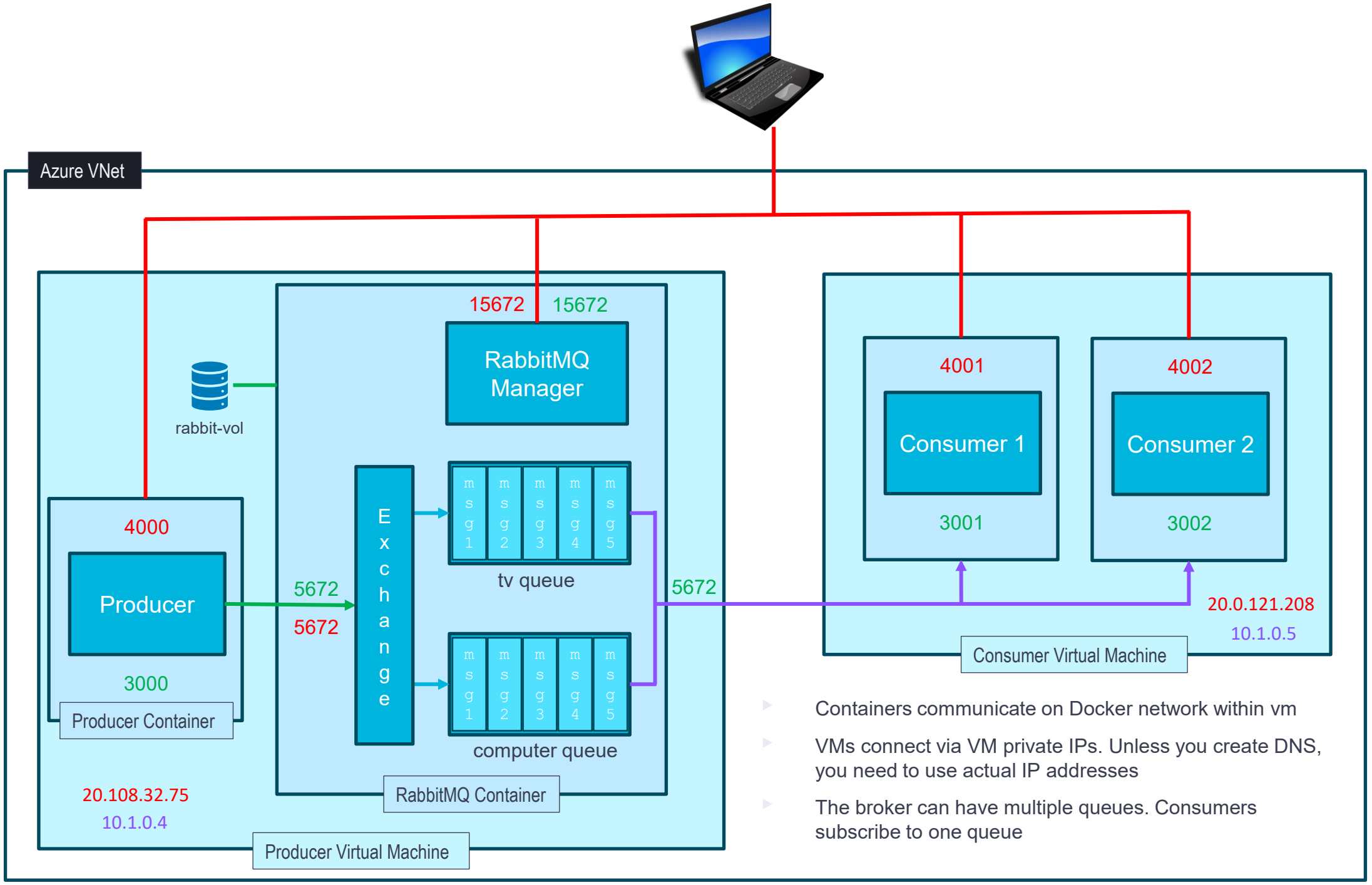
# Rabbitmq exchange types

- Direct
  - a producer and consumer are using one queue based on the binding key (we use this & default exchange)
- Topic
  - a wildcard match between the routing key and the routing pattern determines the queue. Like the Preston example
- Fanout
  - routes a message to all the queues - e.g. message sent to multiple microservices - pub / sub pattern

- All this is summarised in the Rabbitmq documentation as:





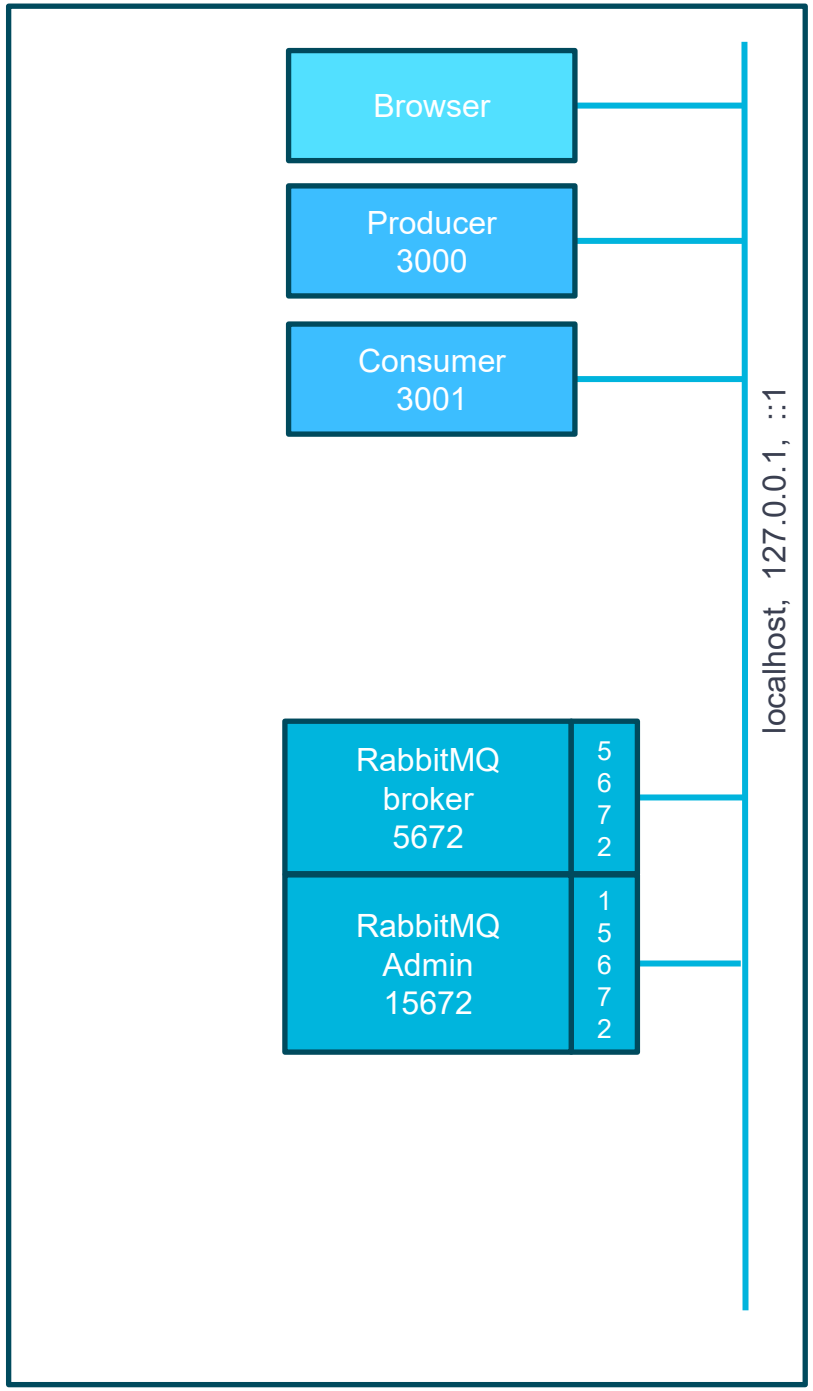

- Containers communicate on Docker network within vm
- VMs connect via VM private IPs. Unless you create DNS, you need to use actual IP addresses
- The broker can have multiple queues. Consumers subscribe to one queue

# RabbitMQ demo

- One VM will host the producer and message broker
- Another will host the consumer

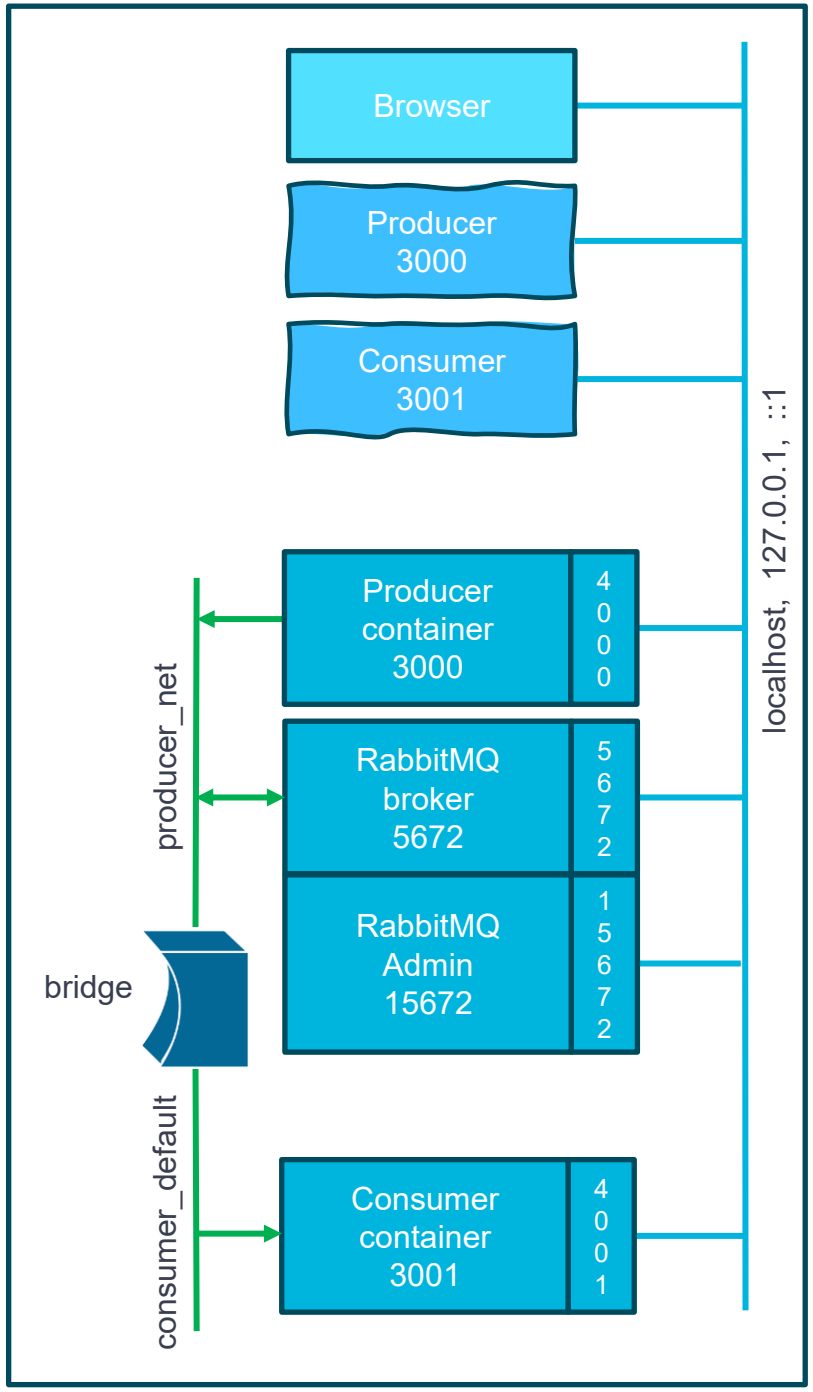
- Test in vscode
  - If consumer is down, producer should still work
  - Start consumer and it should clear the queue

Demo in vscode
\demos\architecture\rabbitOneQueue\

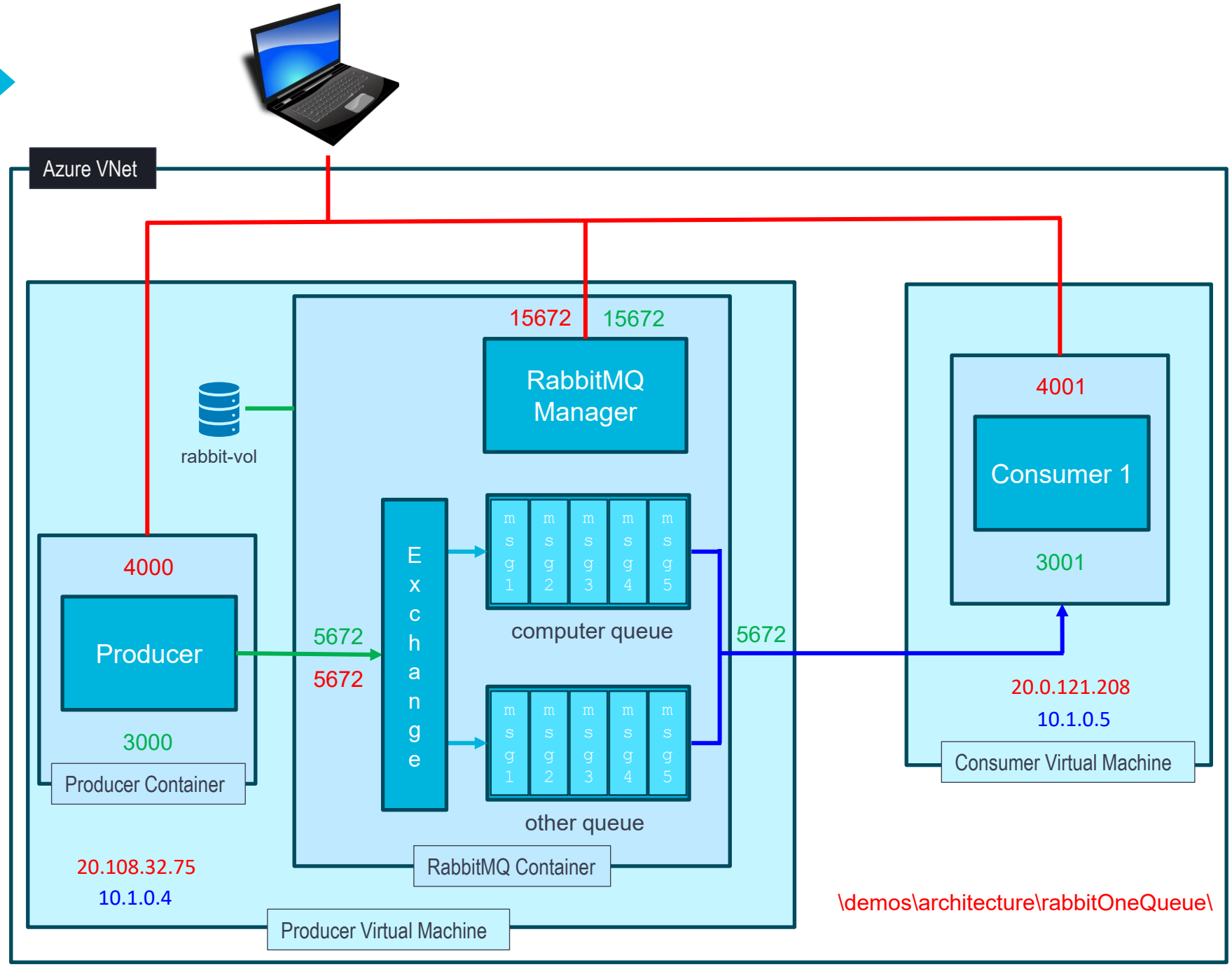


# RabbitMQ demo

- Create two sets of containers as we will deploy on different VMs
- As they are on different networks, we need to join them to test
  - If container recreated, you need to rejoin to network
  - **docker network ls** to list them
  - **docker network connect producer_net consumer**
  - **docker network connect consumer_default producer**
  - **docker network inspect** *network_name* - look in container list
- Deploy images to registry
- Create two VMs and install Docker
  - Remember to change consumer IP to producer private address
- Pull images and start containers
- Test from public IPs




\demos\architecture\rabbitOneQueue\




- Containers communicate on Docker network within vm
- VMs connect via VM private IPs. Unless you create DNS, you need to use actual IP addresses
- VMs need to be in the same VNet
- Producer connects to the broker on port 5672 & registers a callback
- It creates one or more queues
- It sends messages to one or more queues
- The producer is on the same vm as the broker
- The consumer on a different vm connects to the broker on port 5672 and subscribes to a queue
- When a message appears on the queue, the broker calls the consumer's callback function to deliver the message
- Consumer acks the message, the broker removes the message from the queue
- You can view the queues by connecting to the broker's console app on port 15672