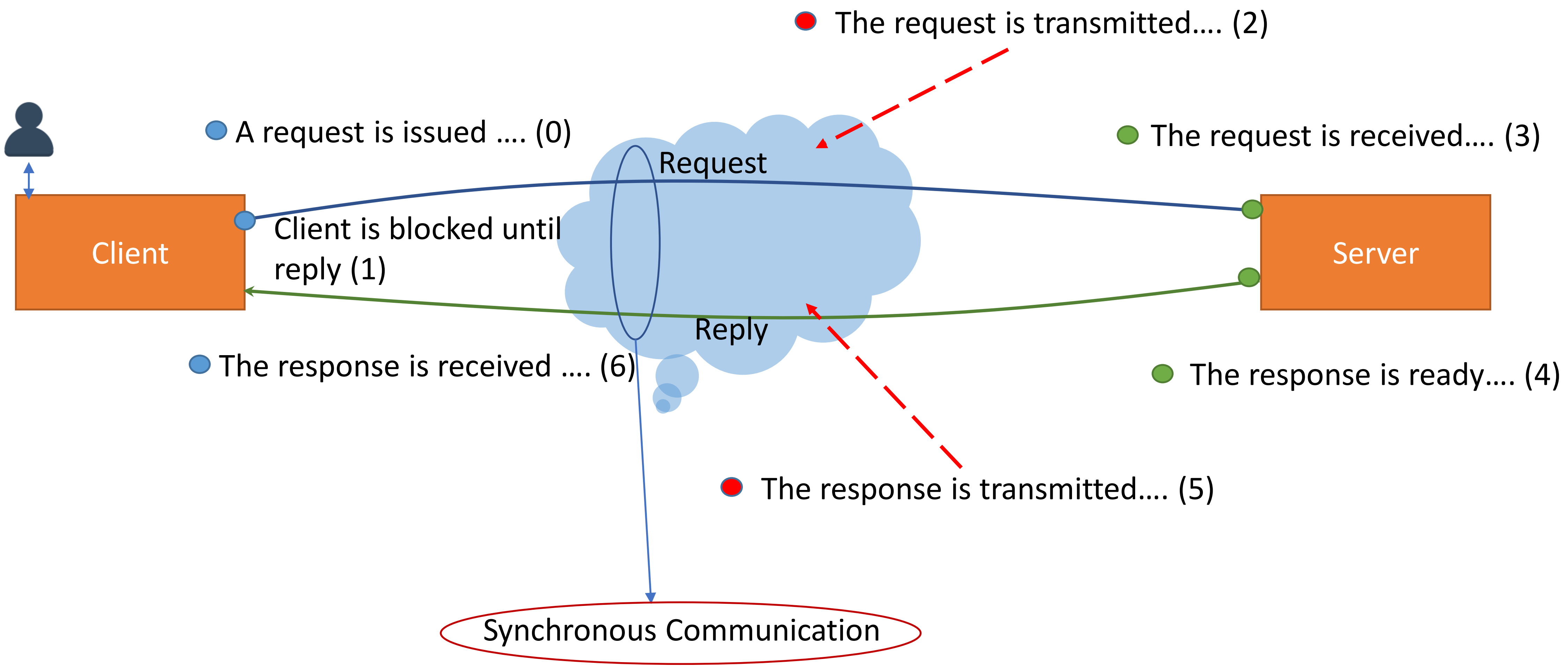
Indirect Communication Using Queues and Web sockets

2021

Contents

- Indirect communication client-server:
    - Point-to-Point communication: Queues
    - Publish-Subscribe communication: Topics
    - Technologies
- Indirect communication server-client:
    - Long pooling
    - WebSocket
- RabbitMQ deployment
- Conclusion

Client-Server Indirect Communication


The request is transmitted…. (2)
A request is issued …. (0)
Request
The request is received…. (3)
Client
Client is blocked until reply (1)
Server
Reply
The response is received …. (6)
The response is ready…. (4)
The response is transmitted…. (5)
Synchronous Communication

- Client waits until server processes request
- Client is blocked until receiving reply
- Sometimes, client does not need the reply – only the ACK that message is being received

Client-Server Indirect Communication

Use cases:

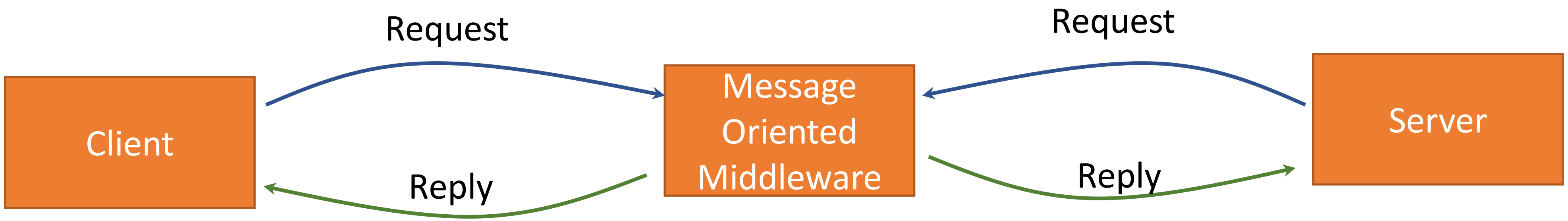
- Applications where server **does not process real time data**
- Applications where **message sending rates** are **variable**

(time intervals with high message rates followed by time intervals with low message rates)

- Applications where there are M data sources and N clients

- Change **transient** communication to **persistent** communication
- Save messages to a data structure
- Send an **ACK** to the client that message is **stored**, and ready to be processed, but **not processed yet**

Some form of Persistent Synchronous Communication

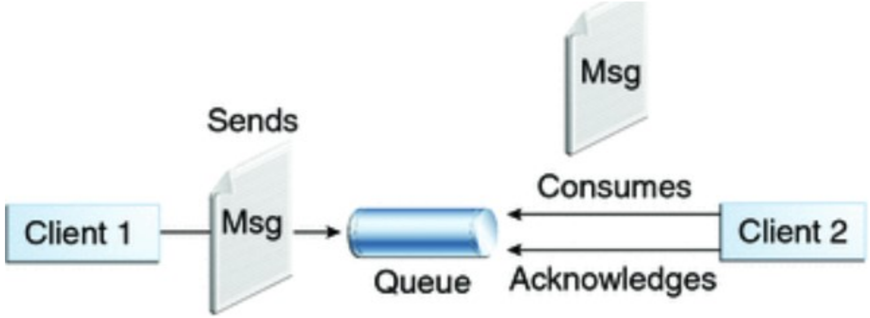


- Create an asynchronous communication based on two synchronous communications and an intermediate entity (Message Oriented Middleware – MOM)

Client-Server Indirect Communication

Message Oriented Middleware Architectures

- **Point-to-point Messaging (Queue destination)**
  - Used when an application needs to send a message to another application
  - The message is first delivered to the queue, and then delivered to a consumer registered for the queue
  - Any number of producers can send messages to the queue
  - Each message is guaranteed to be delivered and consumed by one consumer
  - If no consumers are registered to consume the messages, the queue holds them until a consumer registers to consume them.

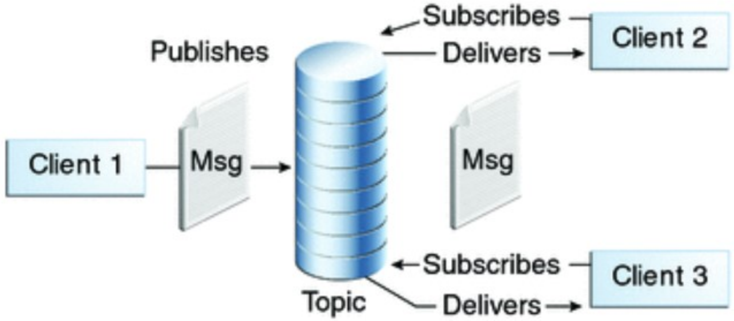


Source: https://docs.oracle.com/javaee/6/tutorial/doc/bncdx.html

Client-Server Indirect Communication

Message Oriented Middleware Architectures

- **Publish – Subscribe Messaging (Topic destination)**
  - Used when multiple applications need to receive the same message
  - Messages are delivered to the topic destination, and then to all active consumers who have *subscribed* to the topic
  - Any number of producers can send messages to a topic destination, and each message can be delivered to any number of subscribers



Source: https://docs.oracle.com/javaee/6/tutorial/doc/bncdx.html

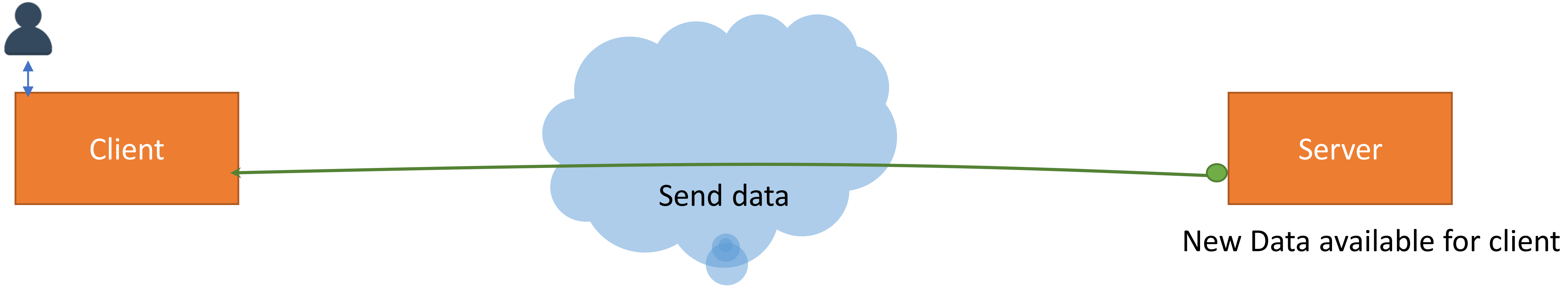
Client-Server Indirect Communication

Technologies:

- **Rabbit MQ**
- JAVA Messaging Service (JMS)
- Microsoft Messaging Queue (MSMQ)
- **Apache Kafka**
- Etc.

Server-Client Indirect Communication

How to handle cases when client needs to be updated without “knowing it”?

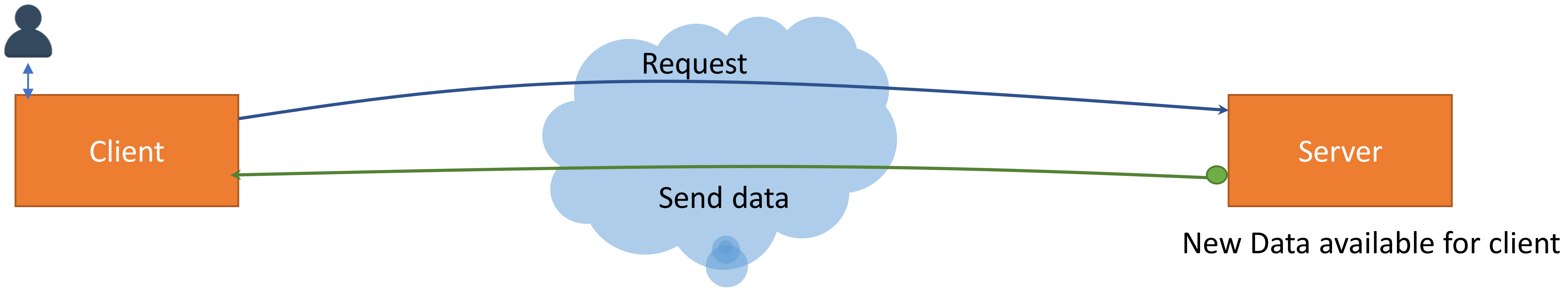


Examples:

- **Chat** – someone writes a message to Client A. Client A is now aware a message was written to it, so it does not know when to make a request to get new messages.
- **News feed** – someone publishes news in a topic of interest. The client would like to receive the news but does not kwon when to ask for it.
- **Sensor monitoring system** – Client visualizes data from a sensor network and is alerted when motion is detected. Client does not know when to ask for data?

Server-Client Indirect Communication

How to handle cases when client needs to be updated without “knowing it”?



Possible Solutions:

- Make repeated HTTP requests based on a timer

→ HTTP was not built to deliver this kind of interactivity

- HTTP is half-duplex => traffic flows in only one direction at a time

Emulating full duplex HTTP

AJAX (Asynchronous JavaScript + XML)

Content can change without loading the entire page

User-perceived low latency

Polling

Nearly real-time

Used in Ajax applications to simulate real-time communication

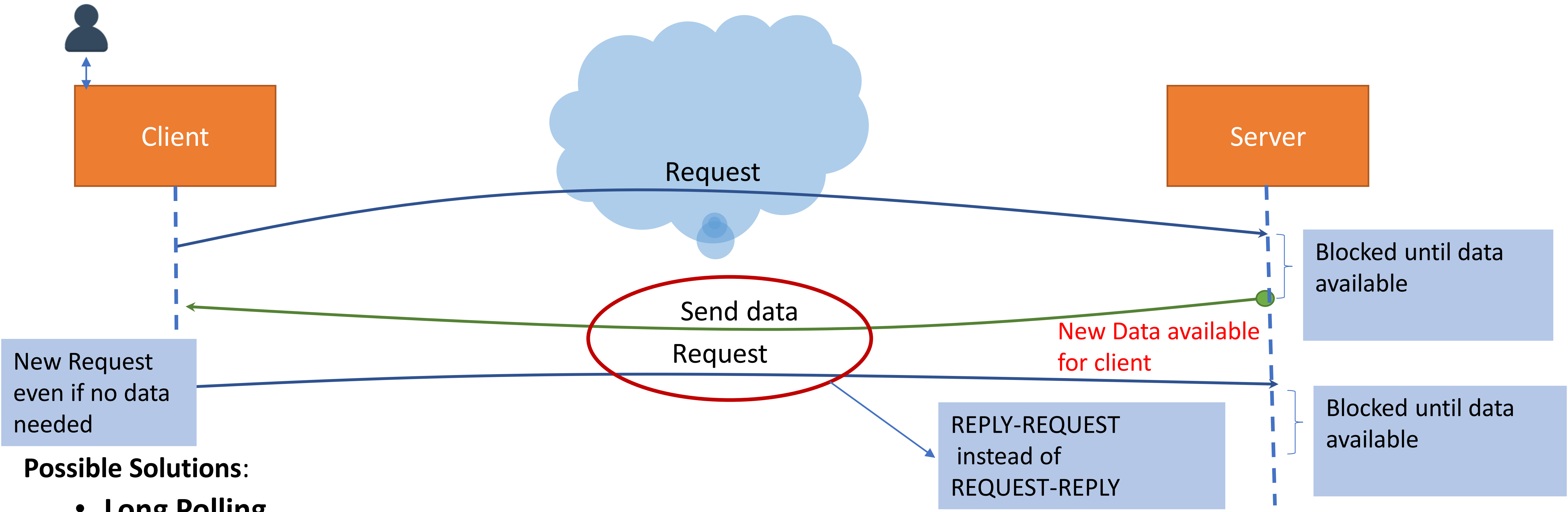
Browser sends HTTP requests at regular time intervals and immediately receives a response

Problem:

- Too many requests – based on frequency of timer
- Some requests do not get new data
- Network congestion
- Server bottleneck

Server-Client Indirect Communication

How to handle cases when client needs to be updated without "knowing it"?



Client

Server

Request

Blocked until data available

Send data

New Data available for client

Request

New Request even if no data needed

REPLY-REQUEST instead of REQUEST-REPLY

Blocked until data available

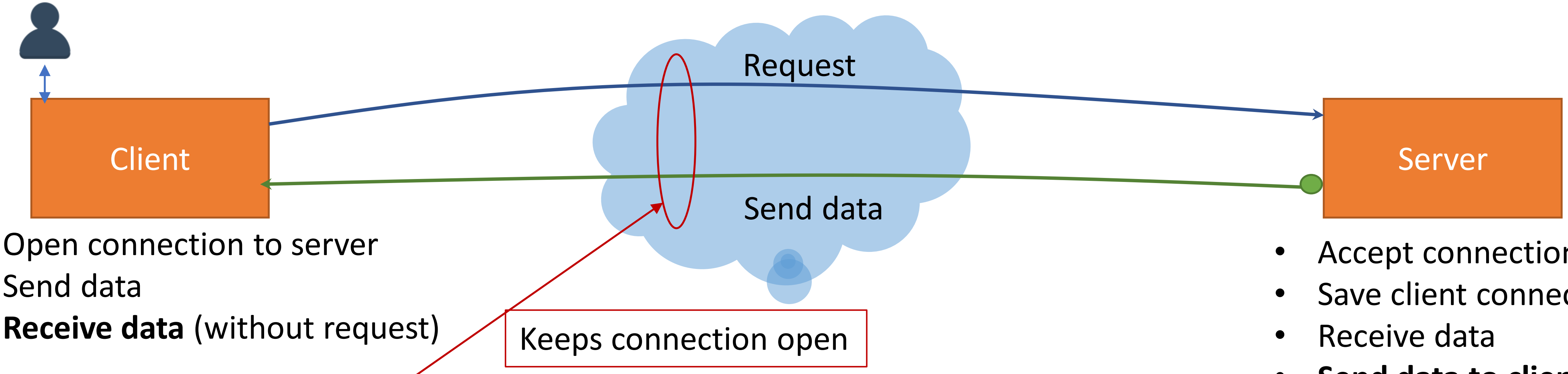
Possible Solutions:

- **Long Polling**
  - usually rely on HTTP => HTTP overhead => less efficient communication between the server and the web browser, especially for real-time applications
  - Browser sends a request to the server which keeps it open for a set period
  - Speed limited by response-request-response
  - Request/response headers add overhead on the wire

Source: https://www.pubnub.com/blog/http-long-polling/

Server-Client Indirect Communication

How to handle cases when client needs to be updated without “knowing it”?



- Open connection to server
- Send data
- **Receive data** (without request)

- Accept connections
- Save client connected
- Receive data
- **Send data to client** (push data)

Possible Solutions:

- **WebSocket:**
  - bi-directional communication (data is sent from the client to the server and vice versa)
  - full-duplex communication (i.e. client and server send data simultaneously) over a single TCP connection
  - real-time communication
  - client/server communication
  - The server can send data to the client at any time
  - Reduces the overhead of each message
  - Uses only one connection per client
    - Opposed to HTTP which creates one request per message

(Source https://docs.spring.io/spring/docs/5.0.0.M5/spring-framework-reference/html/websocket.html)

Conclusion

- Based on asynchronous communication type identified, initiator of data generation, processing type and volume of data exchanged, choose between:

- Asynchronous communication using Message Oriented Middleware:
  - Allows delay tolerant data processing
  - Data generation and processing rates are different in time
  - Queue vs Topic

- Data push techniques:
  - Allows real time data processing
  - WebSocket – mostly used nowadays
  - Based on JavaScript frameworks

Option 1: RabbitMQ – Deployment using Docker on Heroku

- Create a Dockerfile starting from a RabbitMQ image (https://hub.docker.com/_/rabbitmq)

- Setup RabbitMQ to start on $PORT (loaded dynamically from $PORT Environmental Variable)
  - The PORT variable is set by the Heroku runtime and incoming requests are to this port
  - RabbitMQ would start by default on port 5672 => you need to change the default configuration
  - Configure the port by specifying your custom configurations through :
    - *rabbitmq.conf* - used to set things like TCP port, SSL certificates
    - *rabbitmq-env.conf* - used to set environment variables that are read upon startup docs

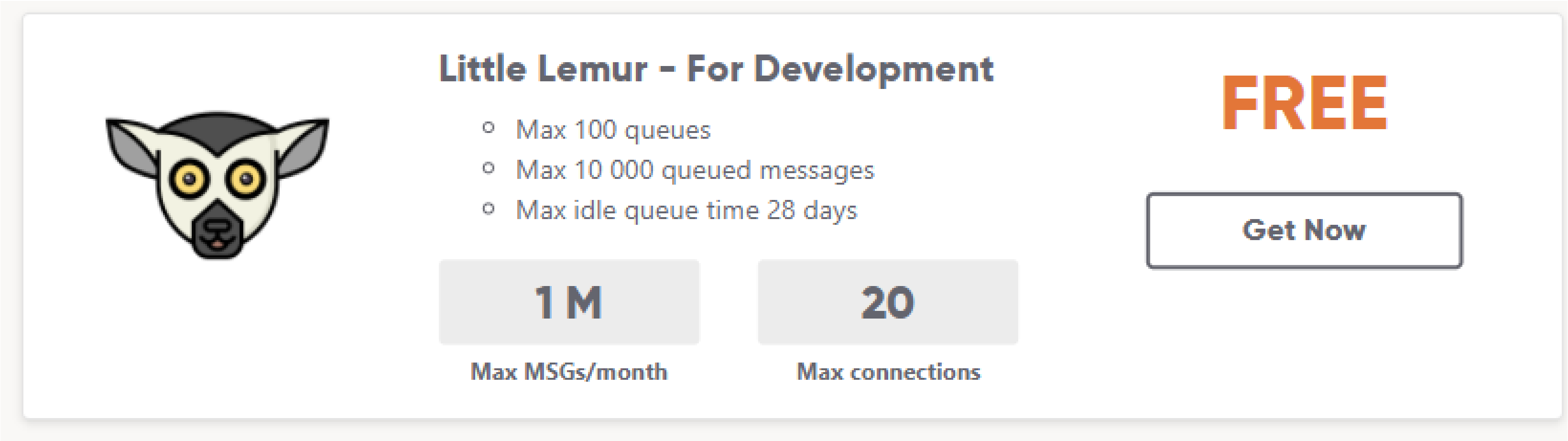
- Deploy your image from your local computer (use *> heroku container:push…)*

Or

- Setup a Gitlab repository with the Dockerfile and a gitlab-ci.yml file

Option 2: RabbitMQ – Use CloudAMQP Free service

- Use RabbitmMQ as a service from https://www.cloudamqp.com/
- Free Plan:



Little Lemur - For Development

- Max 100 queues
- Max 10 000 queued messages
- Max idle queue time 28 days

1 M	20
Max MSGs/month	Max connections

FREE