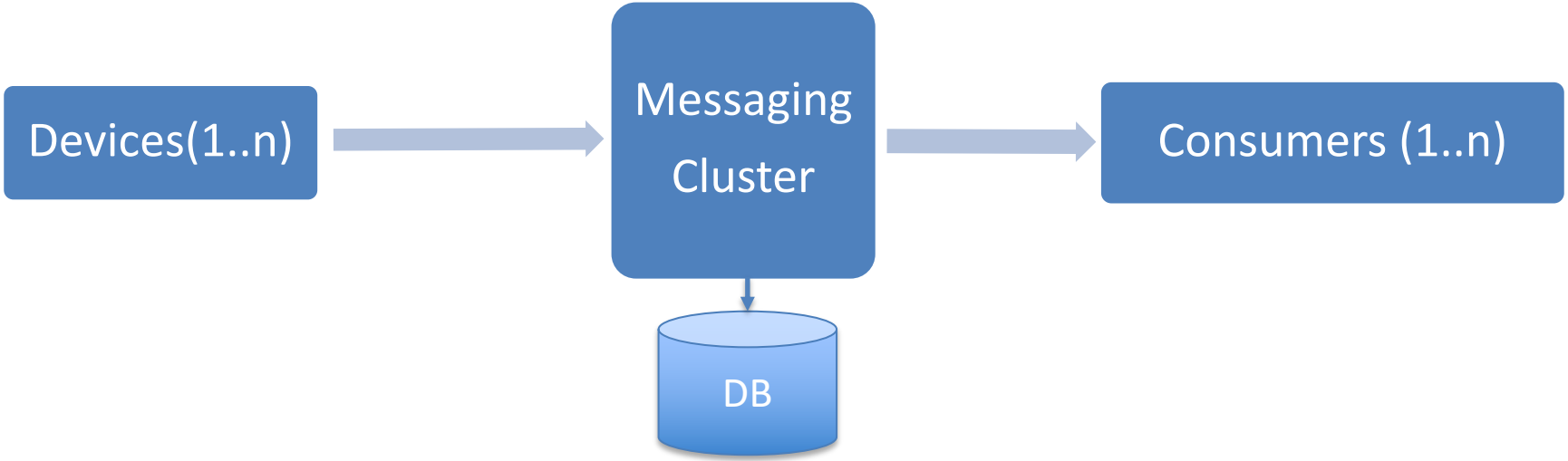
Streaming data use case

Andy Shi Alibaba
Andy.shi@Alibaba-inc.com
Twitter: @andyshi

Use Case : Log aggregation



Challenge : Send log files on demand

1. Debug log file on individual device
2. Locate the device and file
3. Pulling not pushing
4. No DB or buffer allowed
5. Support large number of devices
6. Multiple consumer can query/search in real time
7. Consumer has flow control (Bonus point)

Complexity using messaging queues

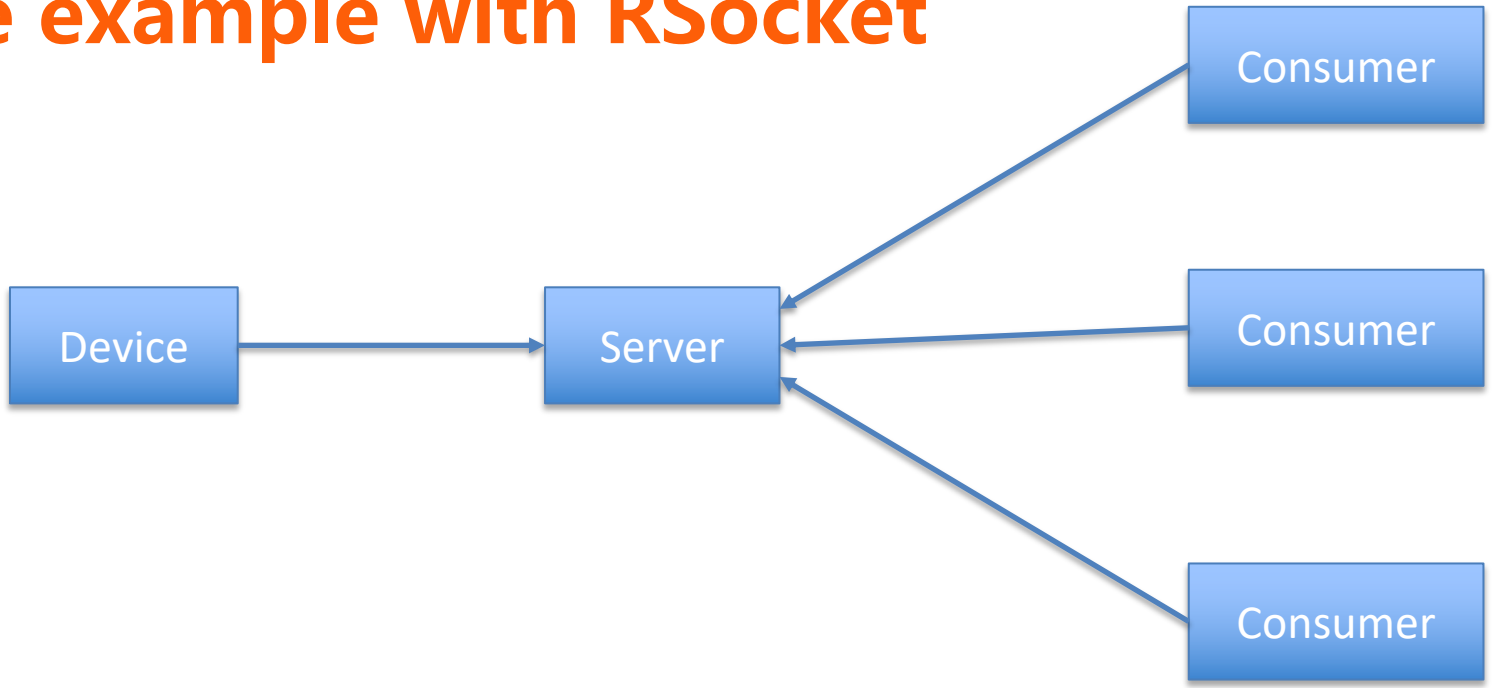
- How many queues for each device?
  - ***Number of APIs x 2***
  - How about file data? ***Number of APIs x 3***
  - How to identify which consumer to send to?
    - ***(Number of APIs x 3)x Numbers of consumers***
  - And repeat that on the consumer side

Illustration of the math problem

- ONLY 2 commands each consumer

RSocket solution

Same example with RSocket



Server/Broker code

- CloseableChannel tcp =
  RSocketFactory.*receive*()
  …
  .transport(TcpServerTransport.*create*(9090))
  …
- CloseableChannel ws =
  RSocketFactory.*receive*()
  …
  .transport(WebsocketServerTransport.*create*("localhost", 9091))

Device code

- DeviceSetupData setupData = new DeviceSetupData(new ArrayList<>(files.keySet()), *clientId*);

- RSocketFactory.*connect*()
  .setupPayload(DefaultPayload.*create*(JsonUtil.*serialize*(setupData)))
  …
  .transport(TcpClientTransport.*create*(9090))

Server/Broker code

- ```
  SocketAcceptor socketAcceptor =
      (setup, sendingSocket) ->
          Mono.fromSupplier(
              () -> {
                DeviceSetupData setupData =
                    JsonUtil.deserialize(setup.getDataUtf8(), DeviceSetupData.class);

                deviceInfoMap.computeIfAbsent(
                    setupData.getDeviceId(),
                    deviceId -> {
                      logger.info("received new incoming connection from device id {}", deviceId);

                      return new DeviceInfo(
                          deviceId,
                          sendingSocket,
                          setupData.getFiles(),
                          () -> {
                            logger.info("device id {} connection closed", deviceId);
                            deviceInfoMap.remove(deviceId);
                          });
                    });
                return new AbstractRSocket() {};
              });
  ```
  ```

Server/Broker code

- ```
  private Flux<Payload> streamSpecificClientFile(String clientId, int
  bufferSize, String file) {
    DeviceInfo deviceInfo = this.clientInfo.get(clientId);
    if (deviceInfo == null) {
      return Flux.empty();
    } else {
      RequestFileStream request = new RequestFileStream(bufferSize, file);

      return deviceInfo
          .getrSocket()
          .requestStream(DefaultPayload.create(JsonUtil.serialize(request)));
    }
  }
  ```
  ```

Device code

- ```
  public Flux<Payload> requestStream(Payload payload) {
    RequestFileStream request =
        JsonUtil.deserialize(payload.getDataUtf8(), RequestFileStream.class);

    logger.info("received request to stream file: " + request.toString());

    String fileName = request.getFile();
    File file = files.get(fileName);
    if (file == null) {
      logger.error("no file named {} found", fileName);
      return Flux.error(new ApplicationErrorException("no file named " + fileName));
    }

    return FileStreamFactory.streamFile(file,
  request.getBufferSize()).map(DefaultPayload::create);
  }
  ```
  ```

Connections needed for one device

- One connection
- Consumers and devices connect independently
- APIs can change without changing streams or connections

Demo

Summary: RSocket simplifies the architecture

- RSocket is connection oriented
- RSocket stream is bi-directional
- RSocket provides end to end flow control