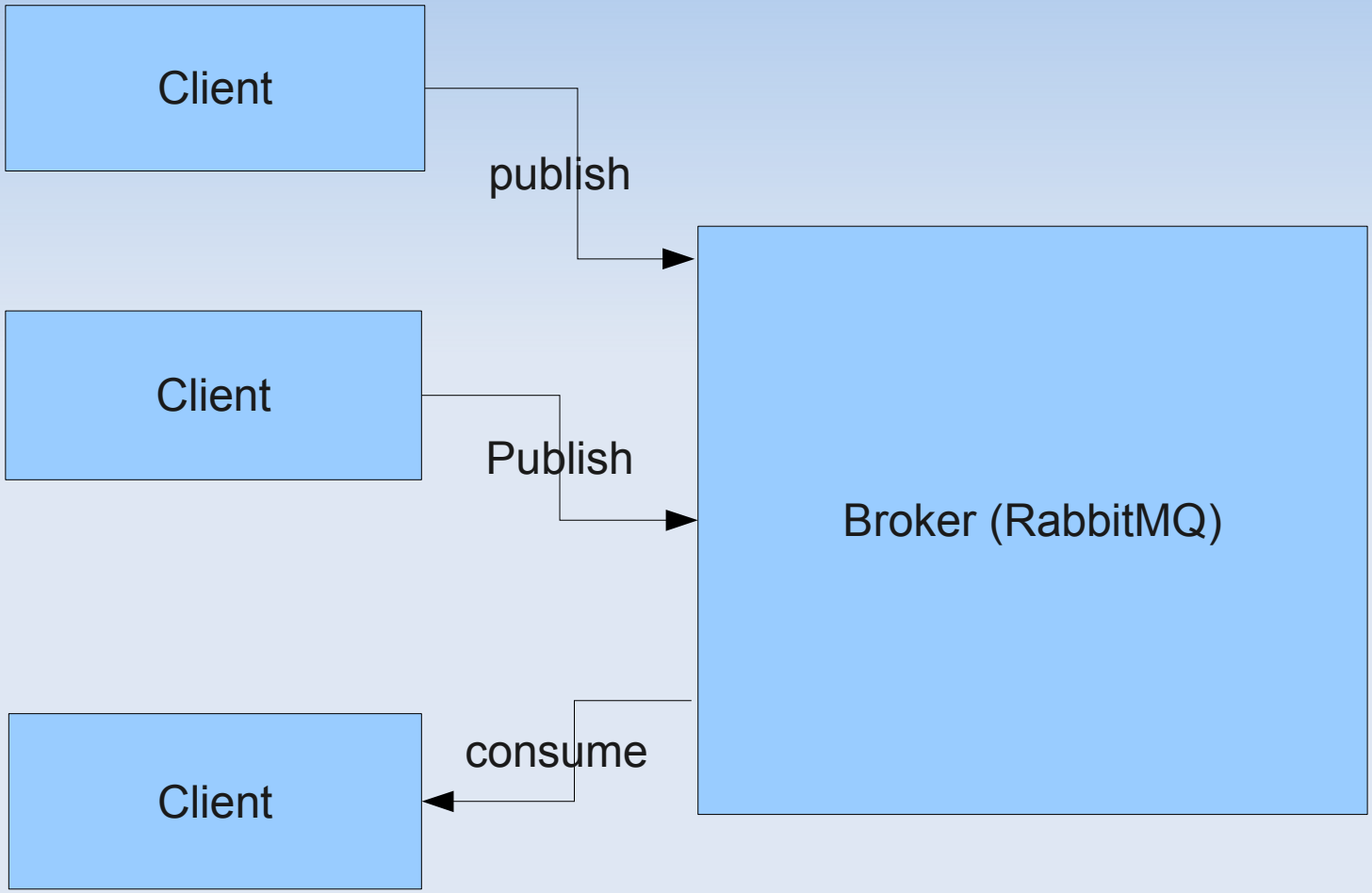
Netamqp

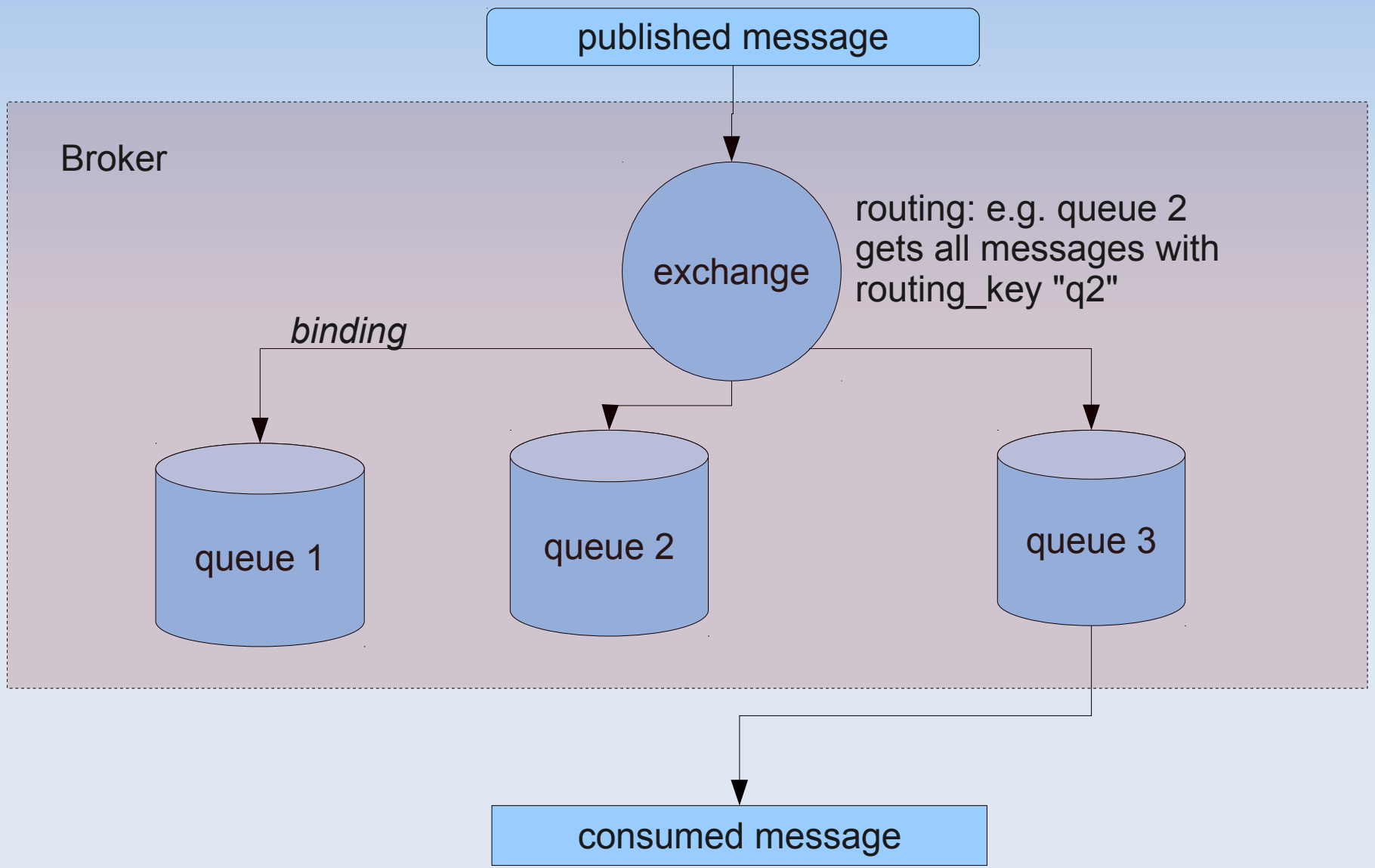
- Architecture
- Connections, Channels, Classes, Methods
- How to open connections and channels
- Exceptions
- Preparing a queue
- Publishing
- Receiving

AMQP architecture



Both publishers and consumers are *clients* of the broker

AMQP message flow



Exchanges

- Predefined exchanges:
  - **amq.fanout:** true (all queues with bindings to this exchange get all messages)
  - **amq.direct:** msg.routing_key = binding.routing_key
  - **amq.topic:** msg.routing_key ~ binding.routing_key (with wildcards)
  - **amq.headers:** routing condition depends on msg headers

Connections and channels

- Connection = TCP connection
- Handshake at beginning of connection, and at end of connection
- Several independent data streams are multiplexed over a single connection. These streams are called **channels**
- Channels are numbered 1-65535. The client chooses the channel number, and has to open the channel. Both peers can close the channel
- Only one activity at a time per channel

Exceptions

- Errors are reported
  - for the connection, or
  - for the channel
- The connection or channel is closed if an error occurs

Classes

- Broker functionality is divided into 6 classes:
  - Connection (→ Netamqp_connection)
  - Channel (→ Netamqp_channel)
  - Exchange (→ Netamqp_exchange)
  - Queue (→ Netamqp_queue)
  - Basic (→ Netamqp_basic)
  - Tx (→ Netamqp_tx)

Methods

- Methods: These are control messages sent via channels
- Methods exist per class
- Example:
  - Channel.open: sent by client to server
  - Channel.open-ok: response by server
- Some methods use this request/response scheme, some methods are unidirectional
- Some methods can carry payload data (content messages), e.g. Basic.publish

Netamqp: open connection

- lib/netamqp/tests/t_connection.ml

```
let esys = Unixqueue.create_unix_event_system()
let p = `TCP(`Inet("localhost", Netamqp_endpoint.default_port))
let ep = Netamqp_endpoint.create p (`AMQP_0_9 `One) esys
let c = Netamqp_connection.create ep
let auth = Netamqp_connection.plain_auth "guest" "guest"
Netamqp_connection.open_s c [ auth ] (`Pref "en_US") "/"
```

- The red statement opens the connection
  - auth: username/password
  - en_US: locale for error messages
  - "/": vhost (names a broker partition)

Netamqp: open channel

- Open the channel:

  let channel = 1

  let co = Netamqp_channel.open_s c channel
- "co" is now a channel object. It is needed for all activities on the channel
- Alternative:

  let co =
      Netamqp_channel.open_next_s c

  (Netamqp chooses a channel number automatically.)

Netamqp: declare a queue

- "declare" means: check that this queue exists in this way, or create a new one. If an incompatible queue is in the way, throw an error

```
let resp_fn =
    Netamqp_queue.declare_s
      ~channel:co
      ~queue:qname                    (* just a string *)
      ()

let resp_qn =
    resp_fn
      ~out:(fun ~queue_name ~message_count ~consumer_count →
                queue_name
            )
      ()
```

Netamqp: bind a queue to an exchange

- We use a pre-defined exchange here (no need to create one)
- Netamqp_queue.bind_s
  ~channel:co
  ~queue:qname
  ~exchange:Netamqp_exchange.amq_direct
  ~routing_key
  ()
- The routing_key is a string that is used by the exchange for message routing

Netamqp: publish 1

- Create content message:

```
let body_string = "this is the payload of the message"

let msg =
      Netamqp_basic.create_message
        (* optional args: *)
        ~content_type:"text/plain"
        ~content_encoding:"ISO-8859-1"
        ~headers: [      "foo", `Longstr "foofield";
                         "bar", `Bool true;
                         "baz", `Sint4 (Rtypes.int4_of_int 0xdd);
                  ]
        ~delivery_mode:1   (* non-persistent *)
        (* this is required: *)
        [Netamqp_rtypes.mk_mstring body_string ]
```

Netamqp: publish 2

- Publish the message:

  Netamqp_basic.publish_s
  ~channel:co
  ~exchange:Netamqp_exchange.amq_direct
  ~routing_key
  msg

- Warning: we do not get feedback about errors during publication (→ use Tx to enable)
- Full example: tests/t_sender_highlevel.ml

Netamqp: consume 1

- How to set up a consumer:
  - Step 1: Define a callback that is invoked for each consumed message
  - Step 2: Enable consumption
  - Step 3: Run the event queue
- Full example:

  tests/t_receiver_highlevel.ml

Netamqp: consume 2

- Step 1:

```
Netamqp_basic.on_deliver
    ~channel:co
    ~cb:(fun ~consumer_tag ~delivery_tag ~redelivered
                ~exchange ~routing_key
                msg →
            …
        )
```

- "msg" is now same object as at publish time
- msg#amqp_body retrieves the body
- msg#content_type retrieves the MIME type
- Refinement of step 1 will be discussed later

Netamqp: consume 3

- Step 2:

  Enable consumption

```
let consumer_tag =
    Netamqp_basic.consume_s
      ~channel:co
      ~queue:qname
      ()
```

- Step 3:

```
Unixqueue.run esys
```

Netamqp: consume 4

- Normally, AMQP requires that we ack each message we consume
  - Can be turned off: ~no_ack:true
  - When there is an unacknowledged message, the broker won't send us more messages
  - Useful when there are several consumers reading from the same queue: The consumers can signal the broker whether they are idle (no un-ack'ed message) or busy (un-ack'ed messages exist)

Netamqp: consume 5

- Step 1, refined:

```
Netamqp_basic.on_deliver
   ~channel:co
   ~cb:(fun ~consumer_tag ~delivery_tag ~redelivered
              ~exchange ~routing_key
             msg →
          (* now process msg, and leave msg un-ack'ed *)
          …
          (* we are done with processing, so ack: *)
           ignore(
             Netamqp_basic.ack_e    (* don't use ack_s here! *)
               ~channel:co
               ~delivery_tag
               ()
           )
       )
```