A C++ MQTT Message Broker for the Enterprise

John Dubchak

Castlight Health Inc.

Agenda

- Shameless self-promotion
- Overview of Messaging
  - Basic messaging concepts
  - Enterprise Messaging
- The MQTT Messaging Protocol
- Designing and Implementing an MQTT Broker

ABOUT ME

The Basics

MESSAGING OVERVIEW

What is Messaging?

The communication of data from sender to receiver over a given messaging channel in an agreed upon format of exchange

What is Messaging?

The **communication** of data from sender to receiver over a given messaging channel in an agreed upon format of exchange

What is Messaging?

The communication of **data** from sender to receiver over a given messaging channel in an agreed upon format of exchange

What is Messaging?

The communication of data from **sender** to receiver over a given messaging channel in an agreed upon format of exchange

What is Messaging?

The communication of data from sender to **receiver** over a given messaging channel in an agreed upon format of exchange

What is Messaging?

The communication of data from sender to receiver over a given **messaging channel** in an agreed upon format of exchange

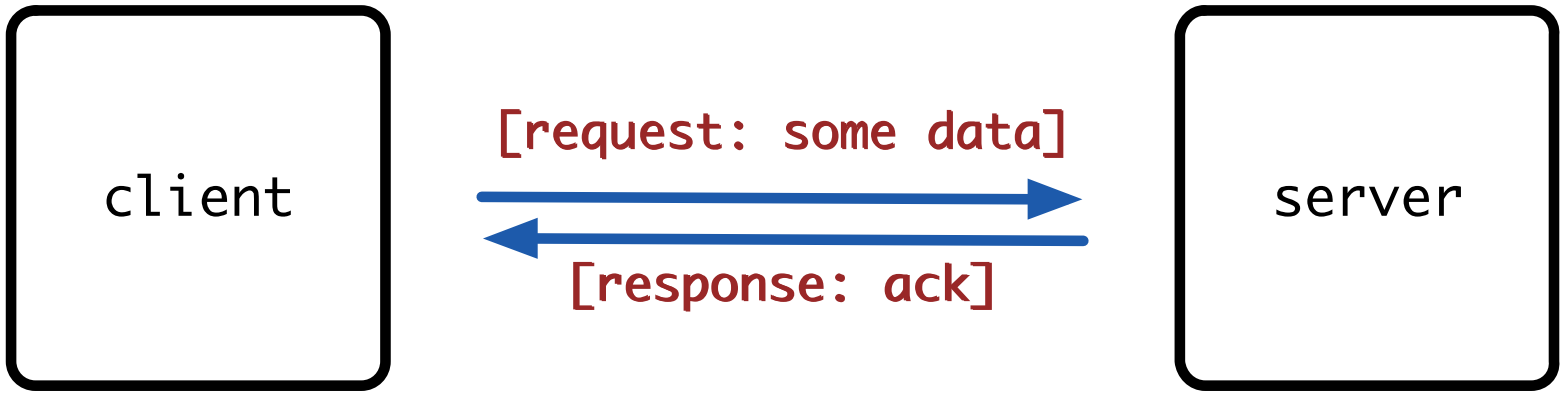
What is Messaging?

The communication of data from sender to receiver over a given messaging channel in an **agreed upon** format of exchange

What is Messaging?

The communication of data from sender to receiver over a given messaging channel in an agreed upon format of **exchange**

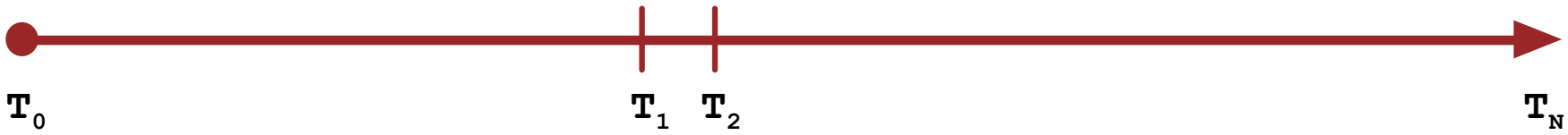
The Canonical Messaging Model



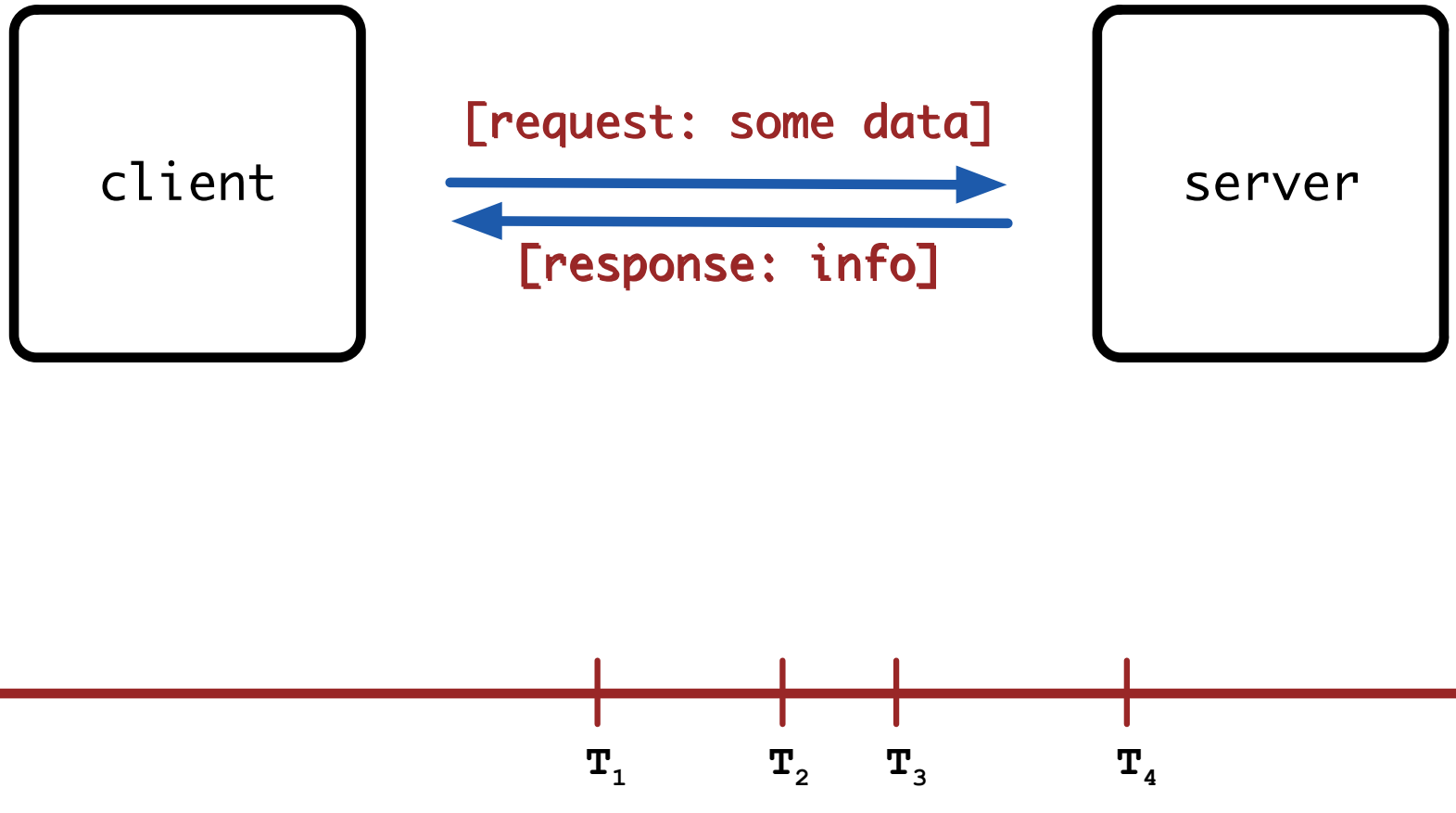
Messaging Architecture

STYLES OF COMMUNICATION

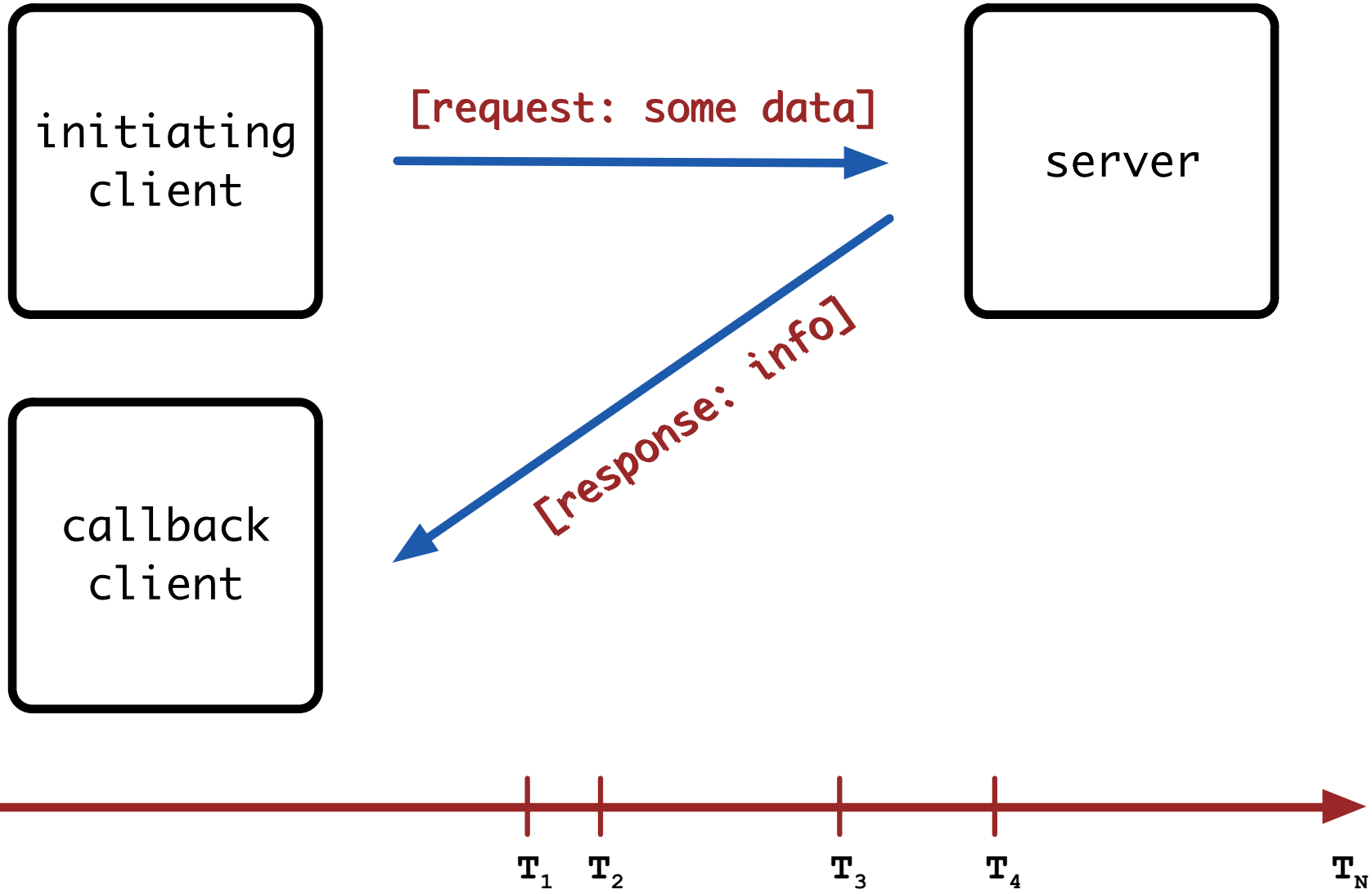
Point-to-Point Messaging



Request-Response



Request-Callback

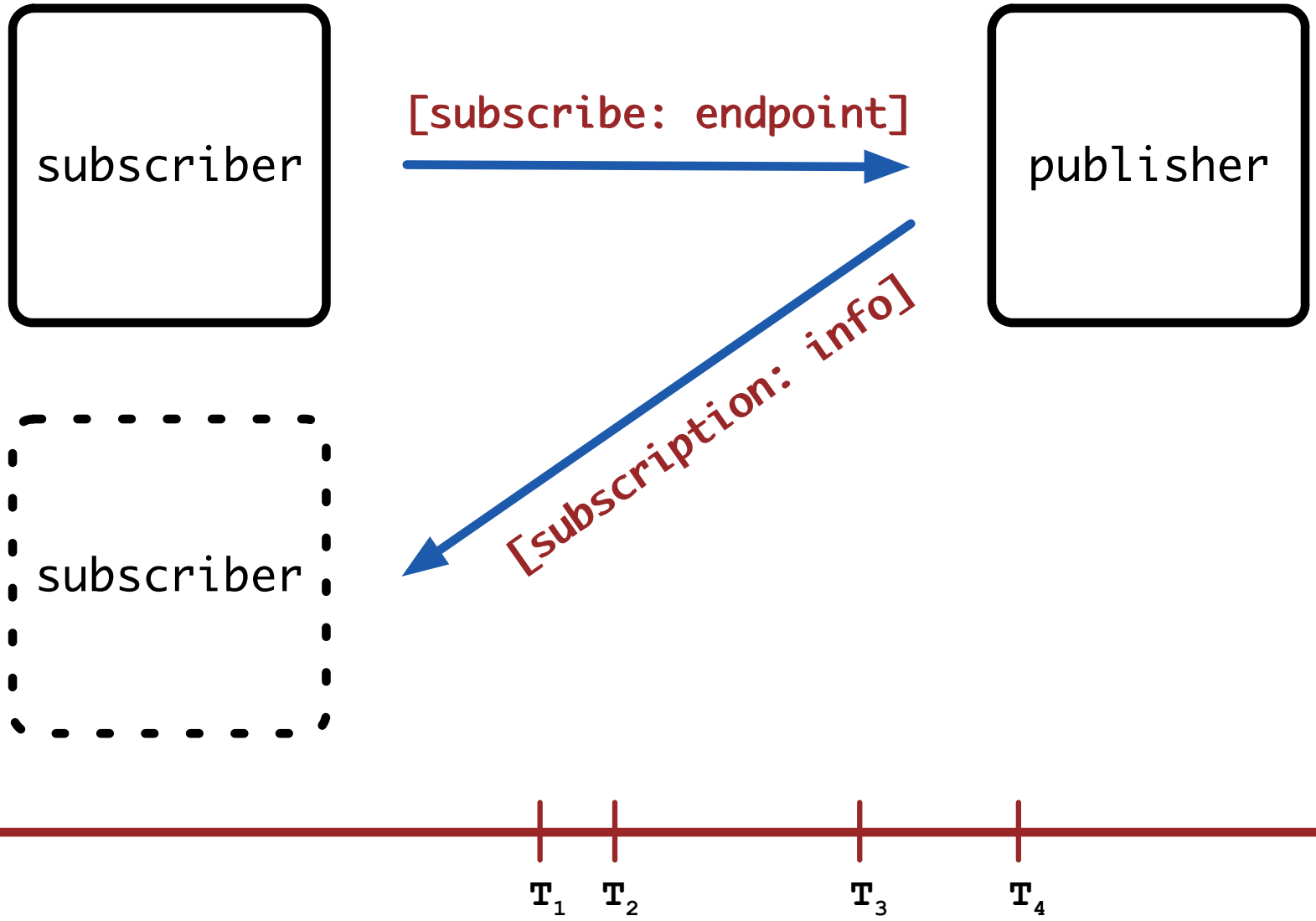


Actor Model

“The actor model in computer science is a mathematical model of concurrent computation that treats ***actors*** as the universal primitives of concurrent computation.”

https://en.wikipedia.org/wiki/Actor_model

Publish-Subscribe



ENTERPRISE MESSAGING

What is Enterprise Messaging?

- A suite of tools providing:
  - Business Process Orchestration
  - Systems and Data Integration
  - Monitoring
  - Transformation/Routing
  - Logging
- You may see MSMQ, IBM MQ Series, JMS Technologies
- Systems are either heterogeneous or homogenous

Architectural Attributes

- Highly Available
- Fault tolerant
- Secure
- Redundant
- Reliable/Delivery Retry
- Guaranteed Message Delivery
- Message Ordering
- Client Session Management

The Messaging Layer

THE MQTT PROTOCOL

What is MQTT?

- Message Queue Telemetry Transport
- Pub/Sub Message Protocol
- Standardized in OASIS & ISO
- Lightweight
- Constrained Vocabulary

Protocol Requirements

- Integer data values are 16-bits in Big-Endian order
- All text is UTF-8 encoded strings
- All strings are prefixed with a two byte length
- Strings are limited to 65,535 bytes
- Character data must be well-formed
- The NULL character ‘\0’ is not permitted in String data

Message Types

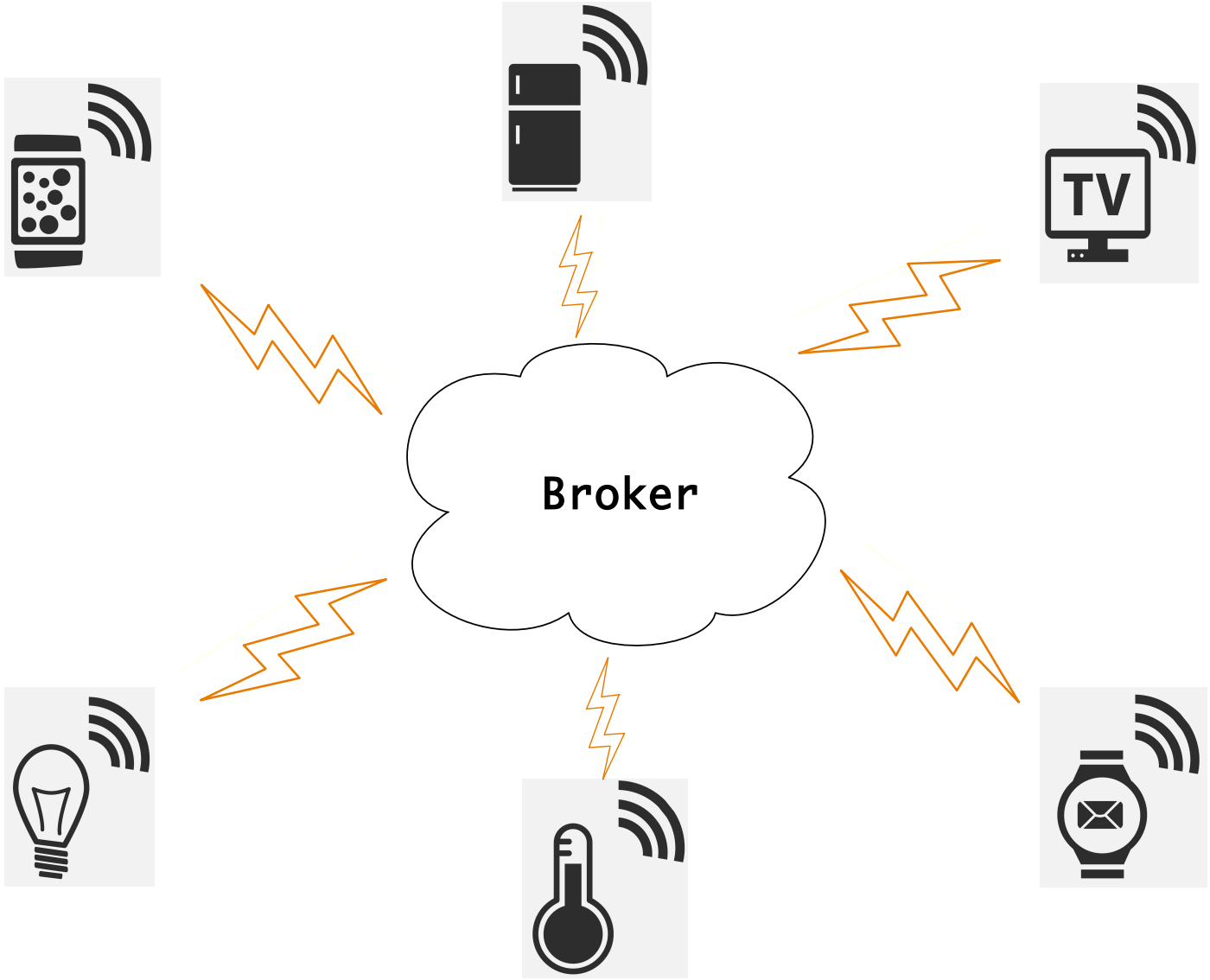
- Connect
- Subscribe
- Unsubscribe
- Publish
- PubRel
- PubComp
- PingReq
- Disconnect

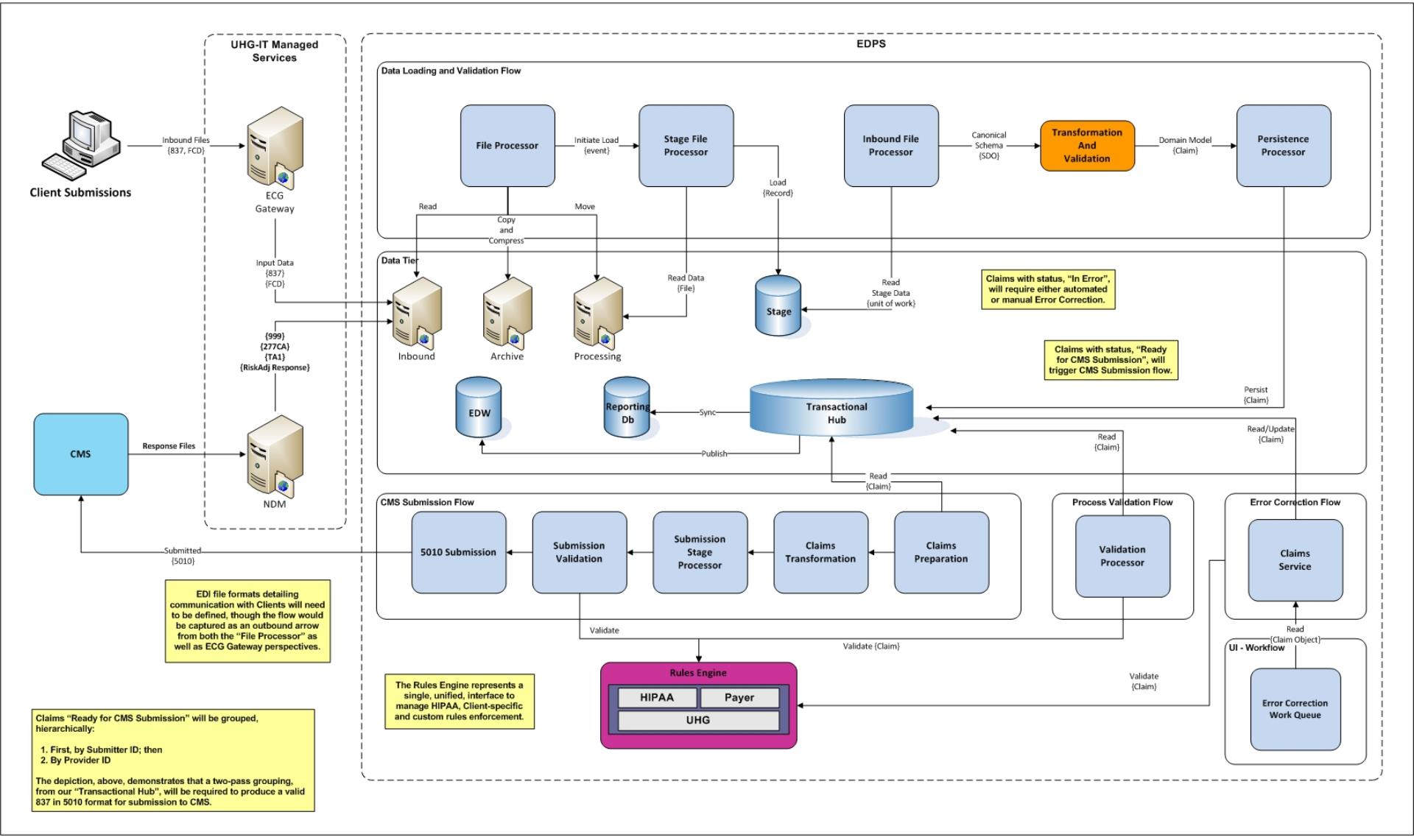
- ConnAck
- Suback
- Unsuback
- Puback

- PingResp

MQTT MESSAGING USE-CASES

Internet of Things

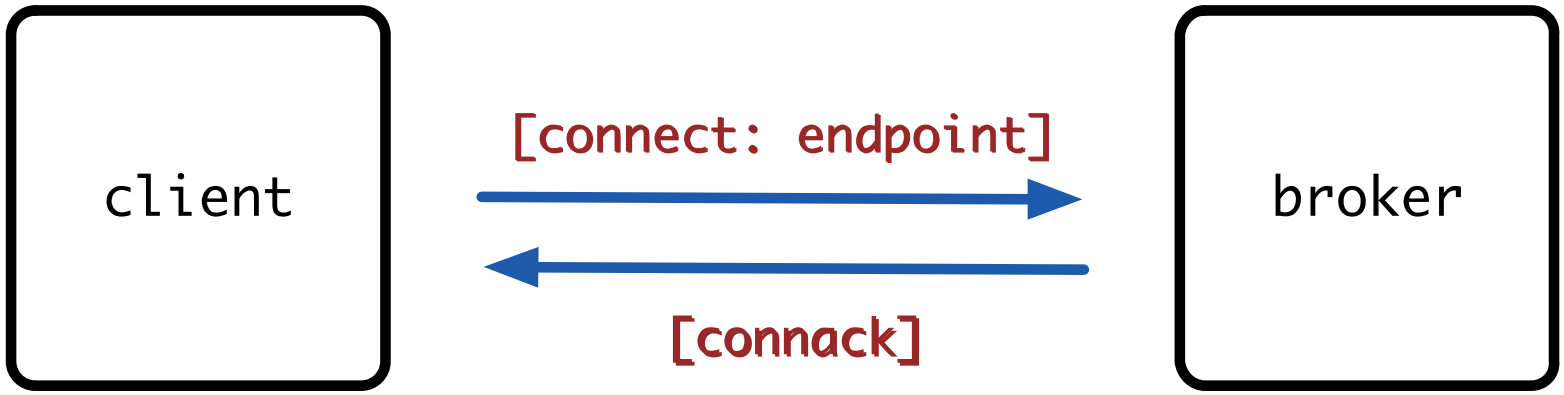



UHG-IT Managed Services
EDPS
Client Submissions
Inbound Files {837, FCD}
ECG Gateway
Input Data {837} {FCD}
{999} {277CA} {TA1} {RiskAdj Response}
NDM
CMS
Response Files
Submitted {5010}
Data Loading and Validation Flow
File Processor
Initiate Load {event}
Stage File Processor
Load {Record}
Inbound File Processor
Canonical Schema {SDO}
Transformation And Validation
Domain Model {Claim}
Persistence Processor
Read
Copy and Compress
Move
Data Tier
Inbound
Archive
Processing
Read Data {File}
Stage
Read Stage Data {unit of work}
Claims with status, "In Error", will require either automated or manual Error Correction.
Claims with status, "Ready for CMS Submission", will trigger CMS Submission flow.
EDW
Reporting Db
Sync
Transactional Hub
Persist {Claim}
Read/Update {Claim}
Read {Claim}
Publish
Read {Claim}
CMS Submission Flow
5010 Submission
Submission Validation
Submission Stage Processor
Claims Transformation
Claims Preparation
Process Validation Flow
Validation Processor
Error Correction Flow
Claims Service
Read {Claim Object}
UI - Workflow
Error Correction Work Queue
Validate
Validate {Claim}
Validate {Claim}
Rules Engine
HIPAA
Payer
UHG
EDI file formats detailing communication with Clients will need to be defined, though the flow would be captured as an outbound arrow from both the "File Processor" as well as ECG Gateway perspectives.
The Rules Engine represents a single, unified, interface to manage HIPAA, Client-specific and custom rules enforcement.
Claims "Ready for CMS Submission" will be grouped, hierarchically:
1. First, by Submitter ID; then
2. By Provider ID
The depiction, above, demonstrates that a two-pass grouping, from our "Transactional Hub", will be required to produce a valid 837 in 5010 format for submission to CMS.

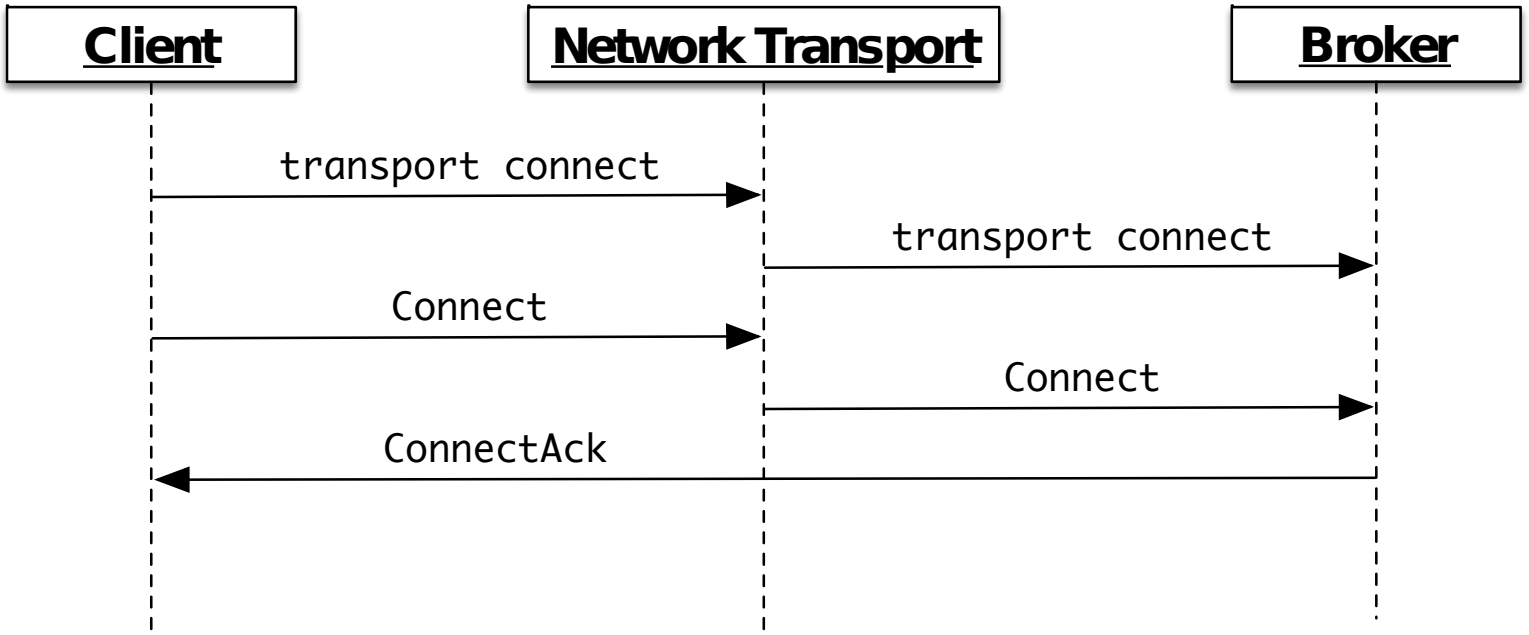
Making a connection

MQTT CLIENT CONNECT TO BROKER

MQTT Connect



MQTT Client Connect



PODs Used for Protocol Connect Data Communications

MQTT ABSTRACTIONS

```
struct Connect {

     std::uint8_t type     : 4;
     std::uint8_t          : 4;
     std::uint8_t length   : 8;

     char* protocol;
     char* level;

     union {
          unsigned char all;
          struct {
               std::uint8_t              : 1;
               std::uint8_t cleanSession : 1;
               std::uint8_t will         : 1;
               std::uint8_t qos          : 2;
               std::uint8_t will_retain  : 1;
               std::uint8_t password     : 1
               std::uint8_t username     : 1;
          } flags;
     } bits;

     char* clientId;
     char* topic;
     char* message;
     char* username;
     char* password;
};
```

```
struct Connect {

     std::uint8_t type     : 4;
     std::uint8_t          : 4;
     std::uint8_t length   : 8;

     char* protocol;
     char* level;

     union {
          unsigned char all;
          struct {
               std::uint8_t              : 1;
               std::uint8_t cleanSession : 1;
               std::uint8_t will         : 1;
               std::uint8_t qos          : 2;
               std::uint8_t will_retain  : 1;
               std::uint8_t password     : 1
               std::uint8_t username     : 1;
          } flags;
     } bits;

     char* clientId;
     char* topic;
     char* message;
     char* username;
     char* password;
};
```

```
struct Connect {

     std::uint8_t type     : 4;
     std::uint8_t          : 4;
     std::uint8_t length   : 8;

     char* protocol;
     char* level;

     union {
          unsigned char all;
          struct {
               std::uint8_t              : 1;
               std::uint8_t cleanSession : 1;
               std::uint8_t will         : 1;
               std::uint8_t qos          : 2;
               std::uint8_t will_retain  : 1;
               std::uint8_t password     : 1
               std::uint8_t username     : 1;
          } flags;
     } bits;

     char* clientId;
     char* topic;
     char* message;
     char* username;
     char* password;
};
```

```
struct Connect {

     std::uint8_t type     : 4;
     std::uint8_t          : 4;
     std::uint8_t length   : 8;

     char* protocol;
     char* level;

     union {
          unsigned char all;
          struct {
               std::uint8_t              : 1;
               std::uint8_t cleanSession : 1;
               std::uint8_t will         : 1;
               std::uint8_t qos          : 2;
               std::uint8_t will_retain  : 1;
               std::uint8_t password     : 1
               std::uint8_t username     : 1;
          } flags;
     } bits;

     char* clientId;
     char* topic;
     char* message;
     char* username;
     char* password;
};
```

```
struct Connect {

     std::uint8_t type     : 4;
     std::uint8_t          : 4;
     std::uint8_t length   : 8;

     char* protocol;
     char* level;

     union {
          unsigned char all;
          struct {
               std::uint8_t              : 1;
               std::uint8_t cleanSession : 1;
               std::uint8_t will         : 1;
               std::uint8_t qos          : 2;
               std::uint8_t will_retain  : 1;
               std::uint8_t password     : 1
               std::uint8_t username     : 1;
          } flags;
     } bits;

     char* clientId;
     char* topic;
     char* message;
     char* username;
     char* password;
};
```

Connection Flags

Flag Name	Purpose
Clean Session	0: MUST resume communication based on current session of Client Identifier
Will	1: If connection is accepted a message must be stored that is delivered upon subsequent disconnection 0: No message is to be stored
Will QoS	If Will Message is 0, Will QoS MUST be 0. If Will Message is 1, Will QoS can be either 0 (0x00), 1 (0x01), or 2 (0x02)
Will Retain	Broker should retain Will Message or not
Password	0, password must not be present, 1 it is
User Name	0, user name must not be present, 1 it is

```
struct ConnAck {

    std::uint8_t type          :
4;
    std::uint8_t               :
4;
    std::uint8_t length        :
8;
    std::uint8_t               :
7;
    std::uint8_t session       :
1;
    std::uint8_t return_code   :
8;
};
```

* Session and return code are only used in ACK

```
struct ConnAck {

    std::uint8_t type          :
4;
    std::uint8_t               :
4;
    std::uint8_t length        :
8;
    std::uint8_t               :
7;
    std::uint8_t session       :
1;
    std::uint8_t return_code   :
8;
};
```

* Session and return code are only used in ACK

CONNACK Return Codes

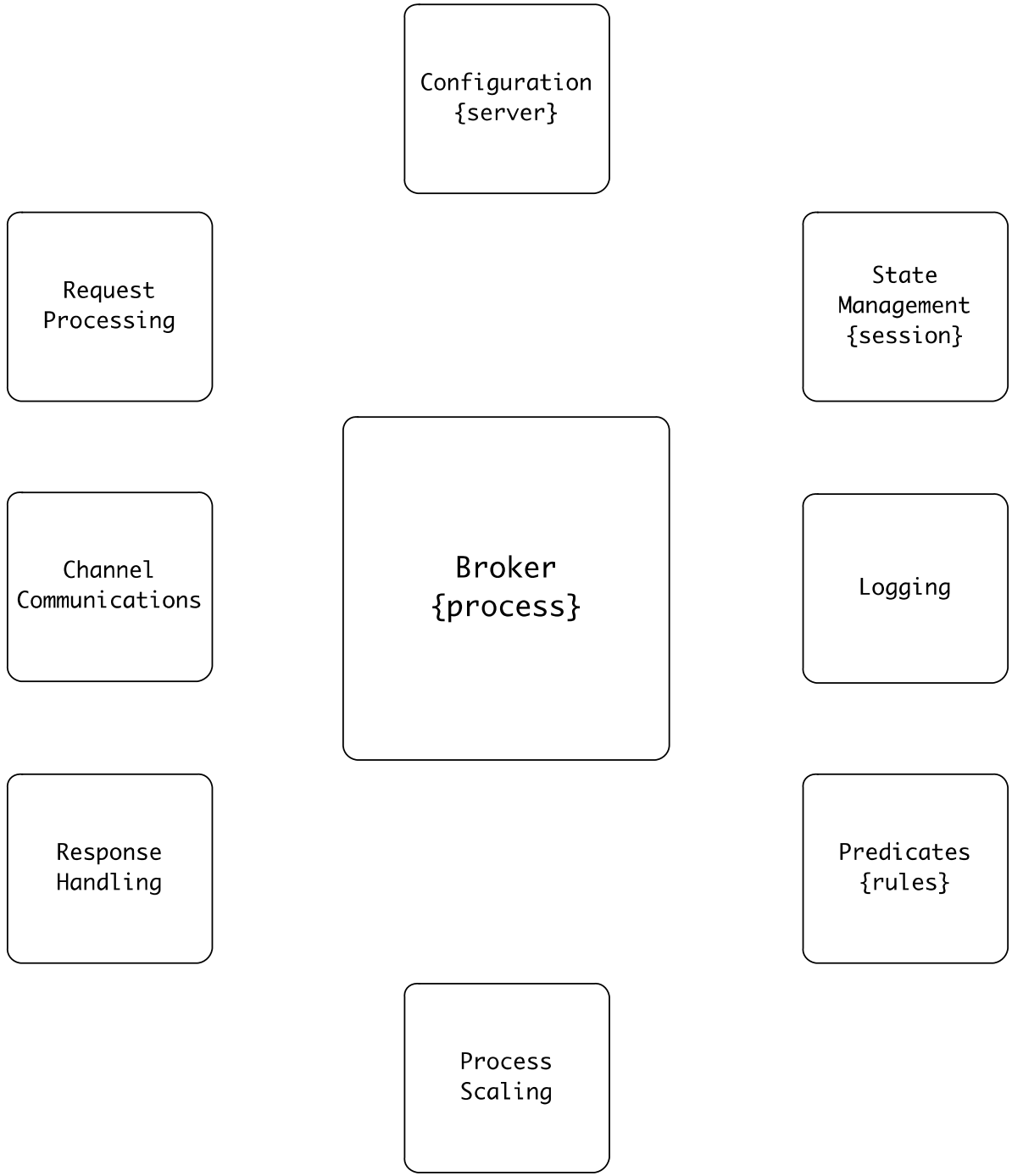
Value	Return Code Response	Description
0	0x00 Connection Accepted	Connection Accepted
1	0x01 Connection Refused	Unsupported MQTT Level
2	0x02 Connection Refused, Identifier Rejected	The Client Identifier is correct UTF-8 but not permitted
3	0x03 Connection Refused, Server Unavailable	The MQTT Service is unavailable
4	0x04 Connection Refused, Bad username or password	Data in the user name or password is malformed
5	0x05 Connection Refused, Not Authorized	Client not authorized to connect
6 – 255		Reserved for future use

Component Design

MQTT BROKER

Preliminaries

- Templates, templates, templates
- Policy-based Design
- Moderate use of Template Metaprogramming
- Implementation based upon First Principles
- Implemented as a daemon on Linux
- Personal research project exploring MQTT


Configuration
{server}
Request
Processing
State
Management
{session}
Broker
{process}
Channel
Communications
Logging
Response
Handling
Predicates
{rules}
Process
Scaling

IMPLEMENTATION DETAILS

Broker

```
template<typename HANDLE,
         template <typename>
                  class Acceptor = Accept>

class MqttBroker : public BasicDaemon
                                <
                                   NetworkService
                                   <


Acceptor<HANDLE>
                                      >
                                >
```

BasicDaemon

```
template
<
  typename T,
  typename DaemonPolicy = daemon_policy,
  typename SigHandlerType = signals::SignalHandler,
  typename DaemonException = std::runtime_error
>
class BasicDaemon
```

Instantiating the Broker

```
// Read configuration etc…

MqttBroker<Handle> server;
server.name(server_name);
server.port(to_int(port));

// Launch the server, bootstrap and
deamonize
server.start();
```

Creating a Linux Daemon

- Fork a process
  - Call fork() - closes parent and makes child parent of init process
  - Call setsid, exit if it returns -1
- Call fork() again – a convenient shortcut
- Clear process' umask
- Change working directory to root directory
- Close all open file descriptors – careful with logging
- Use dup2 to open descriptors 0, 1, and 2 to /dev/null

**** The Linux Programming Interface, Michael Kerrisk***

NetworkService

```
template<typename Accept,
         typename Socket = SocketLifecycle<
                                typename

Accept::ServiceDescriptor>>
struct NetworkService : public Socket, Accept {

    void start() {
        if (!is_listening()) {
            this->connect();
            this->listen();
            this->accept();
        }
    }
};
```

Waiting for Incoming Connections

```
// Back in the broker...
void _accept() override {

  while (true) {
    auto h = this→receive();
    if (this->is_good(h)) {

      if (!this->enable_read(h)) {
        log(str("Cannot continue: ") +
                          ErrorAdapter::get(errno));
      }
      // ... magic ...
    }
  }
}
```

Connection Received

```
auto buffer = handler.handle(h);
auto strategy =

ResponseStrategy<char>::create(
                          get_type(buffer), buffer);
auto exchange =
mqtt::make_exchange(mqtt::make_message<char>
                            (buffer.get()));

exchange.on_status_change(*this);
if (this→config_enabled(exchange)) {
   Task t(concurrent_work, exchange);
   _pool.submit(std::move(t));
}
```

What is an Exchange?

```
enum class ExchangeState {
    CREATED, RUNNING, STOPPED,
    BLOCKED, CANCELED, FINISHED
};

struct Exchange {

    virtual void proceed() = 0;
    virtual ExchangeState status() = 0;
    virtual void status(ExchangeState) = 0;
    virtual Configuration configuration() = 0;
    virtual void configuration(Configuration) =
0;

};
```

An Exchange

- Task starts
- Exchange State:
  - Transitioned from CREATED to INPROGRESS
  - Event callback occurs on Broker
- Broker receives non-const reference to Exchange and can inspect for Client Id, enforce rules, and change state, e.g. block, stop or cancel
  - Depending on Exchange type, Broker may do one of several things
- Upon successful Exchange, final state is FINISHED
- Exchange communicates with Client and

Questions?