# Coroutine & Event Dispatcher

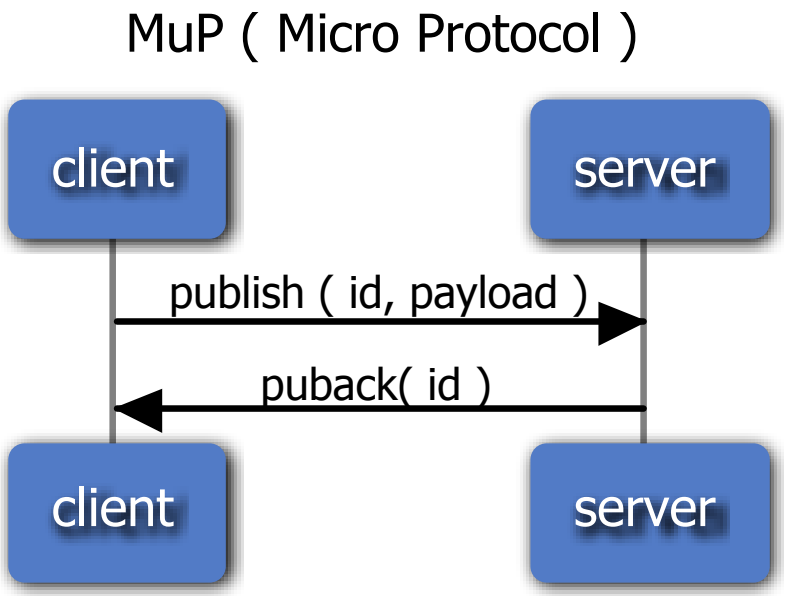
## Non blocking operations in LibXively

Olgierd Humeńczuk

December 13, 2014

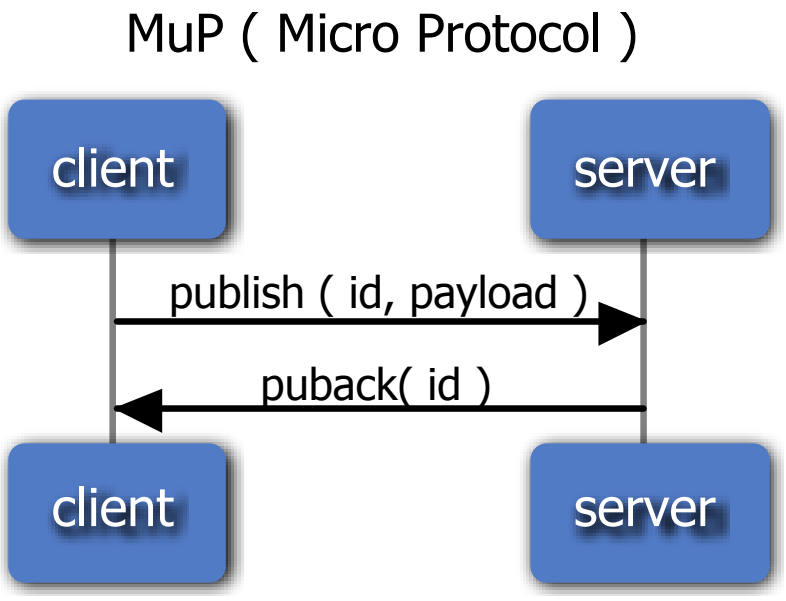
# Brief Outline

# Protocol



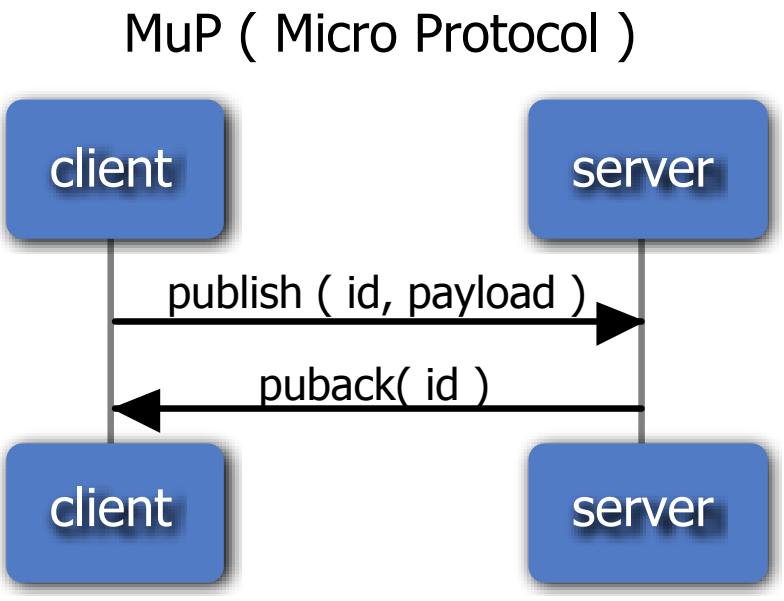

- Let's create simple protocol
- Simple enough to send payloads
- But powerfull enough to make sure it's delivered at least once
- Yes it's a small fraction of MQTT

# Protocol



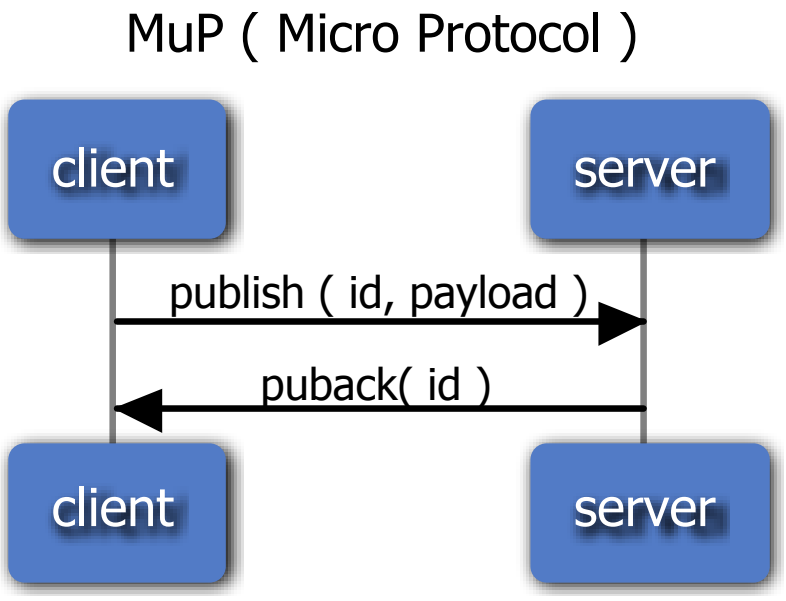

- Let's create simple protocol
- Simple enough to send payloads
- But powerfull enough to make sure it's delivered at least once
- Yes it's a small fraction of MQTT

# Protocol




- Let's create simple protocol
- Simple enough to send payloads
- But powerfull enough to make sure it's delivered at least once
- Yes it's a small fraction of MQTT

# Protocol




- Let's create simple protocol
- Simple enough to send payloads
- But powerfull enough to make sure it's delivered at least once
- Yes it's a small fraction of MQTT

# Definition

## Header

```
typedef enum
{
   M_UNKNOWN = 0, M_PUBLISH, M_PUBACK
} M_message_types;

typedef struct
{
   M_message_types type;
} M_header_t;
```

## Publish/Puback

```
typedef struct
{
   uint16_t id;
   uint8_t* data; uint16_t length;
}M_publish_t;

typedef struct
{
   uint16_t id;
}M_puback_t;
```

## Final Structure

```
typedef struct
{
   M_header_t header;
   union
   {
      M_publish_t publish;
      M_puback_t  puback;
   } types;
} M_protocol_t;
```

# Definition

## Header

```
typedef enum
{
   M_UNKNOWN = 0, M_PUBLISH, M_PUBACK
} M_message_types;

typedef struct
{
   M_message_types type;
} M_header_t;
```

## Publish/Puback

```
typedef struct
{
   uint16_t id;
   uint8_t* data; uint16_t length;
}M_publish_t;

typedef struct
{
   uint16_t id;
}M_puback_t;
```

## Final Structure

```
typedef struct
{
   M_header_t header;
   union
   {
      M_publish_t publish;
      M_puback_t  puback;
   } types;
} M_protocol_t;
```

# Definition

## Header

```
typedef enum
{
   M_UNKNOWN = 0, M_PUBLISH, M_PUBACK
} M_message_types;

typedef struct
{
   M_message_types type;
} M_header_t;
```

## Publish/Puback

```
typedef struct
{
   uint16_t id;
   uint8_t* data; uint16_t length;
}M_publish_t;

typedef struct
{
   uint16_t id;
}M_puback_t;
```

## Final Structure

```
typedef struct
{
   M_header_t header;
   union
   {
      M_publish_t publish;
      M_puback_t  puback;
   } types;
} M_protocol_t;
```

# Sending

## Common

```
void handle_events( buffer_t* data, event_t e, uint32_t time_diff )
{
   static state_t state = STATE_SEND_DATA;
   static uint16_t sent = 0;
   static uint32_t timer_value = 0;
   static bool parser_not_done = false;

   timer_value += time_diff;

   switch( state )
   {
      case STATE_INIT:
      {
         state             = STATE_SEND_DATA;
         sent              = 0;
         parser_not_done   = false;
      }
      break;
```

# Sending

## Sending

```
case STATE_SEND_DATA:
{
   if( e == CAN_WRITE )
   {
      while( sent < data->length )
      {
         int res = write( data->data + sent, data->length - sent );

         if( res <= 0 )
         {
            // do the error handling and maybe leave
         }

         sent += res;
      }

      state       = STATE_RECV_DATA;
      timer_value = 0;
   }
}
break;
```

# Receiving

## Receiving

```
case STATE_RECV_DATA:
{
   if( recvd_msg == NULL ) { ALLOC_AT( M_protocol_t, recvd_msg ); }

   if( e == CAN_READ )
   {
      if( timer_value > TIMEOUT ) { // timeout!!! }

      while( parser_not_done == true )
      {
         ALLOC( buffer_t, recv_data );
         ALLOC_AT( recv_data->data, 32 );

         int res = read( recv_data->data, 32 );
         if( res <= 0 ) { // do the error handling }

         recv_data->length             = res;
         parser_result_t parser_result = call_parser( recv_data
            , recvd_msg
            , &parser_not_done );

         if( parser_result == PARSER_ERROR ) { // do the error handling }
         FREE( recv_data->data );
         FREE( recv_data );
      }
   }

   state = ANALYSE;
}
break;
```

# Receiving

## Analyse

```
    case ANALYSE:
       {
          if( recvd_msg->header.type != M_PUBACK ) { // wrong message type }

          // take the payload

          // send it or read it

          FREE( recvd_msg );
       }
       break;
    };
}
```

# Conclusions

- Code is long and less readable
- This is error prone
  - Memory leaks
  - Reordering sequence in wrong way
  - Level of complication
- Order of states may not reflect the real order of execution

# Conclusions

- Code is long and less readable
- This is error prone
  - Memory leaks
  - Reordering sequence in wrong way
  - Level of complication
- Order of states may not reflect the real order of execution

# Conclusions

- Code is long and less readable
- This is error prone
  - Memory leaks
  - Reordering sequence in wrong way
  - Level of complication
- Order of states may not reflect the real order of execution

# Conclusions

- Code is long and less readable
- This is error prone
  - Memory leaks
  - Reordering sequence in wrong way
  - Level of complication
- Order of states may not reflect the real order of execution

# Conclusions

- Code is long and less readable
- This is error prone
  - Memory leaks
  - Reordering sequence in wrong way
  - Level of complication
- Order of states may not reflect the real order of execution

# Conclusions

- Code is long and less readable
- This is error prone
  - Memory leaks
  - Reordering sequence in wrong way
  - Level of complication
- Order of states may not reflect the real order of execution

# Implementation

## Coroutine macros

```
#define BEGIN_CORO( state )\
    switch( state )\
    { \
        default:

#define YIELD( state, ret )\
    state = __LINE__; return ret; case __LINE__:

#define YIELD_ON( state, expression, ret )\
{ \
    if ( (expression) ) \
    { \
      state = __LINE__; return ret; case __LINE__:; \
    } \
};

#define YIELD_UNTIL( state, expression, ret )\
{ \
    if ( (expression) ) \
    { \
      state = __LINE__; return ret; case __LINE__:; \
      continue; \
    } \
};

#define CORO_END()\
    };
```

## Cons

- Can't use switch inside coroutine
- Requires passing the coroutine state
- Requires to declare all variables before the coroutine
- Requires to re-initialize all variables on every re-entry

# Implementation

## Coroutine macros

```
#define BEGIN_CORO( state )\
    switch( state )\
    { \
        default:

#define YIELD( state, ret )\
    state = __LINE__; return ret; case __LINE__:

#define YIELD_ON( state, expression, ret )\
{ \
    if ( (expression) ) \
    { \
      state = __LINE__; return ret; case __LINE__:; \
    } \
};

#define YIELD_UNTIL( state, expression, ret )\
{ \
    if ( (expression) ) \
    { \
      state = __LINE__; return ret; case __LINE__:; \
      continue; \
    } \
};

#define CORO_END()\
    };
```

## Cons

- Can't use switch inside coroutine
- Requires passing the coroutine state
- Requires to declare all variables before the coroutine
- Requires to re-initialize all variables on every re-entry

# Implementation

## Coroutine macros

```
#define BEGIN_CORO( state )\
    switch( state )\
    { \
        default:

#define YIELD( state, ret )\
    state = __LINE__; return ret; case __LINE__:

#define YIELD_ON( state, expression, ret )\
{ \
    if ( (expression) ) \
    { \
      state = __LINE__; return ret; case __LINE__:; \
    } \
};

#define YIELD_UNTIL( state, expression, ret )\
{ \
    if ( (expression) ) \
    { \
      state = __LINE__; return ret; case __LINE__:; \
      continue; \
    } \
};

#define CORO_END()\
    };
```

## Cons

- Can't use switch inside coroutine
- Requires passing the coroutine state
- Requires to declare all variables before the coroutine
- Requires to re-initialize all variables on every re-entry

# Implementation

## Coroutine macros

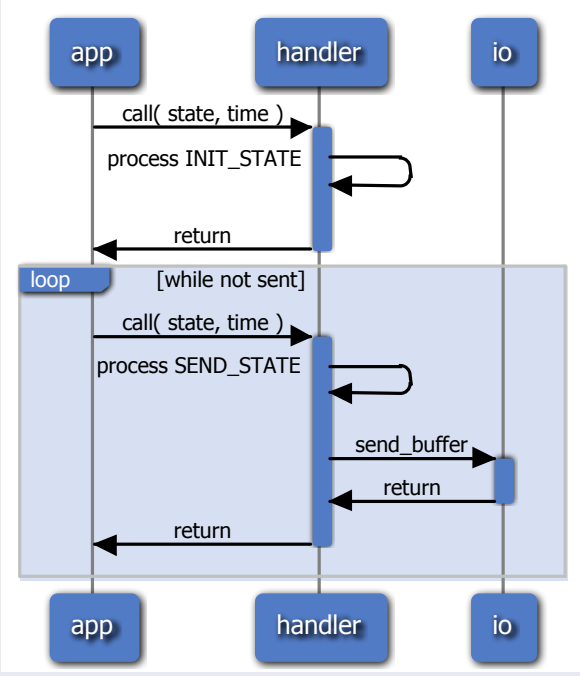
```
#define BEGIN_CORO( state )\
    switch( state )\
    { \
        default:

#define YIELD( state, ret )\
    state = __LINE__; return ret; case __LINE__:

#define YIELD_ON( state, expression, ret )\
{ \
    if ( (expression) ) \
    { \
      state = __LINE__; return ret; case __LINE__:; \
    } \
};

#define YIELD_UNTIL( state, expression, ret )\
{ \
    if ( (expression) ) \
    { \
      state = __LINE__; return ret; case __LINE__:; \
      continue; \
    } \
};

#define CORO_END()\
    };
```

## Cons

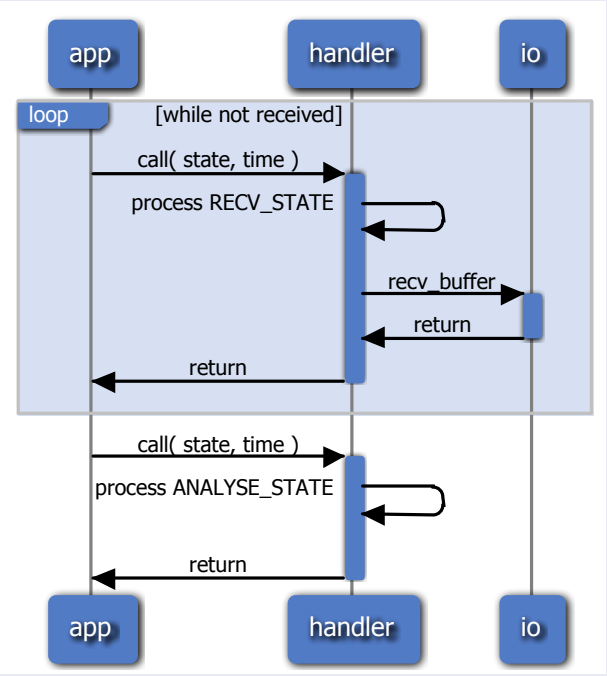
- Can't use switch inside coroutine
- Requires passing the coroutine state
- Requires to declare all variables before the coroutine
- Requires to re-initialize all variables on every re-entry

# Sequence of operations for sending/receiving MuP

## Init and send



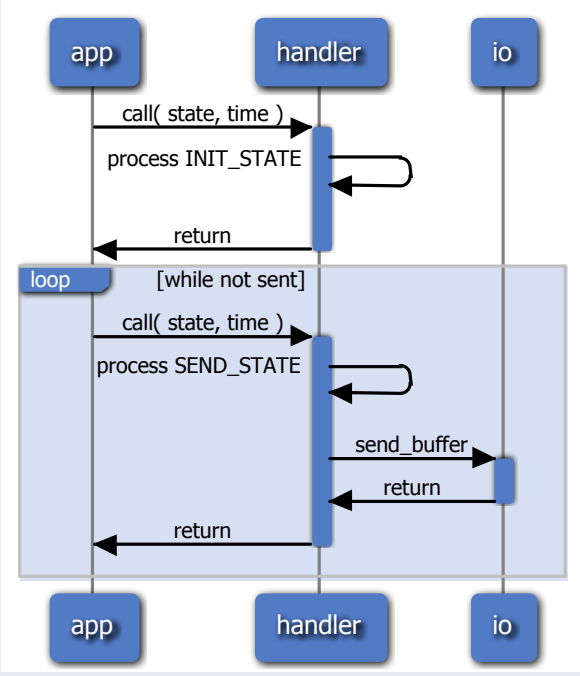

## Receive and analyse



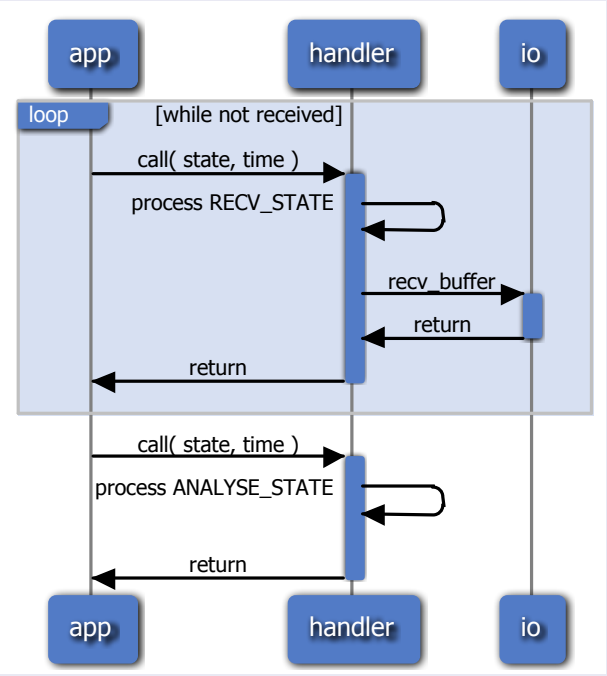

- We are able to execute some part of the code but than we have to wait for the next iteration
- We prereserve the state of processing between each execution
- We enter to the proper place using switch

# Sequence of operations for sending/receiving MuP

## Init and send



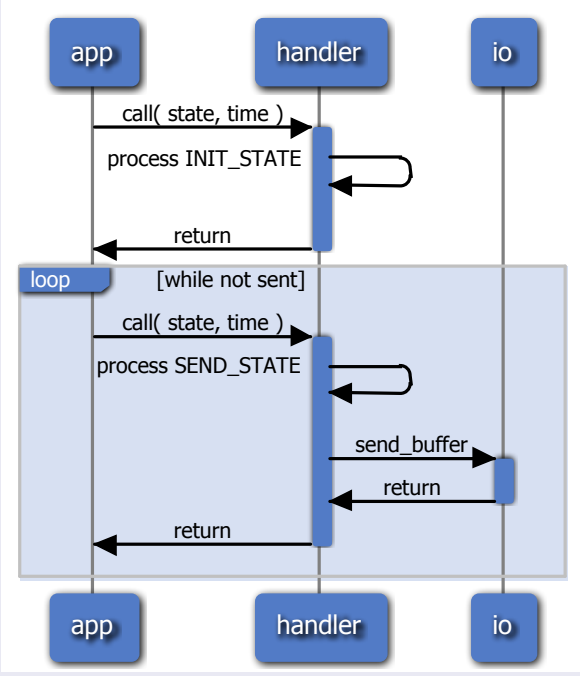

## Receive and analyse



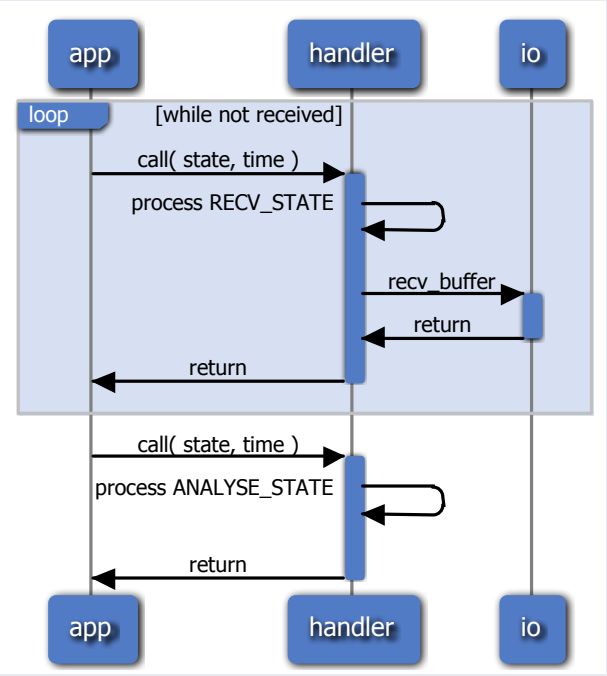

- We are able to execute some part of the code but than we have to wait for the next iteration
- We prereserve the state of processing between each execution
- We enter to the proper place using switch

# Sequence of operations for sending/receiving MuP

## Init and send




## Receive and analyse




- We are able to execute some part of the code but than we have to wait for the next iteration
- We prereserve the state of processing between each execution
- We enter to the proper place using switch

# Solution ?

## Coroutine

```
requested_state_t process( handler_state_t state )
{
   BEGIN_CORO( state->co );

   ALLOC_AT( message_t, state->data );
   ALLOC_BUFFER_AT( message_t, state->data->data );

   do
   {
      res = write( state->data + state->offset, 32 );
      if( res <= 0 ) { // do the error handling and maybe YIELD( state->co ); }
      YIELD_UNTIL( state->co, state->sent < data->lenght, WANT_WRITE )
   } while( state->sent < data->length );

   FREE( state->data->data ); FREE( state->data );

   state->timeout = REGISTER_TIMER( make_handle( process, TIMEOUT_STATE ), TIMEOUT );
   YIELD( state->co, WANT_READ );

   ALLOC_AT( buffer_t, state->buffer ); ALLOC_BUFFER_AT( state->buffer->data, 32 );
   ALLOC_AT( message_t, state->recvd_msg );

   if( state == TIMEOUT ) { //timeout }
   UNREGISTER_TIMER( state->timeout );

   do
   {
      res = read( state->buffer->data, 32 );
      if( res <= 0 ) { // do the error handling and maybe YIELD( state->co ); }
      state->buffer->length = res;
      state->parser_done = parse( state->buffer, state->recvd_msg );
      YIELD_UNTIL( state->co, state->parser_done, WANT_READ );
   } while( state->parser_done );

   FREE( state->buffer->data ); FREE( state->buffer );

   if( recvd_msg->header.type != M_PUBACK ) { // wrong message type }
   // take the payload
   // send it or read it
   FREE( recvd_msg );
   END_CORO();
}
```

# Coroutine analysis - sending

## sending

```
requested_state_t process( handler_state_t state )
{
   BEGIN_CORO( state->co );

   ALLOC_AT( message_t, state->data );
   ALLOC_BUFFER_AT( message_t, state->data->data );

   do
   {
      res = write( state->data + state->offset, 32 );
      if( res <= 0 ) { // do the error handling
                       // and maybe YIELD( state->co ); }
      YIELD_UNTIL( state->co
        , state->sent < data->lenght
        , WANT_WRITE )
   } while( state->sent < data->length );
```

# Coroutine analysis - timeout

## timeout

```
FREE( state->data->data ); FREE( state->data );

state->timeout = REGISTER_TIMER(
     make_handle( process, TIMEOUT_STATE )
     , TIMEOUT );
YIELD( state->co, WANT_READ );
```

# Coroutine analysis - receiving

## receiving

```
ALLOC_AT( buffer_t, state->buffer );
ALLOC_BUFFER_AT( state->buffer->data, 32 );
ALLOC_AT( message_t, state->recvd_msg );

if( state == TIMEOUT ) { //timeout }
UNREGISTER_TIMER( state->timeout );

do
{
   res = read( state->buffer->data, 32 );
   if( res <= 0 ) { // do the error handling
                    //and maybe YIELD( state->co ); }
   state->buffer->length = res;
   state->parser_done = parse( state->buffer, state->recvd_msg );
   YIELD_UNTIL( state->co, state->parser_done, WANT_READ );
} while( state->parser_done );
```

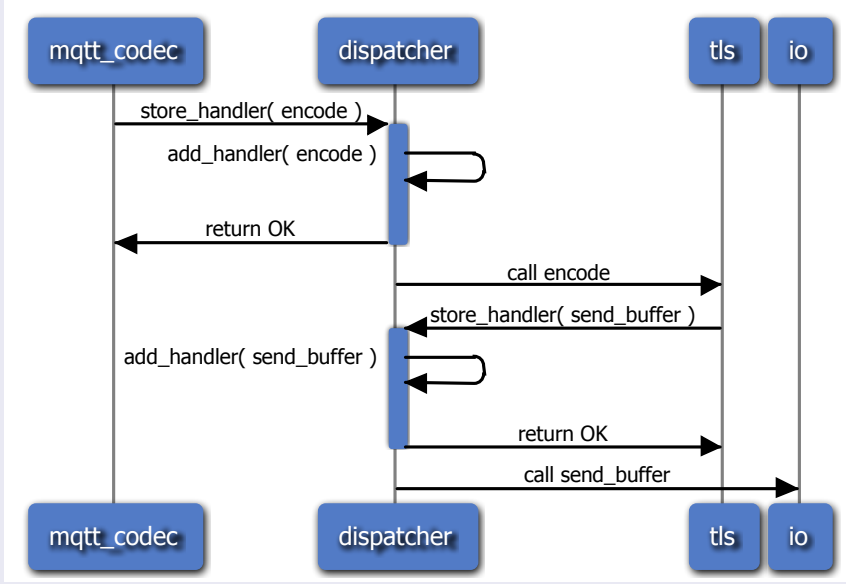
# Coroutine analysis - analyse

## analyse

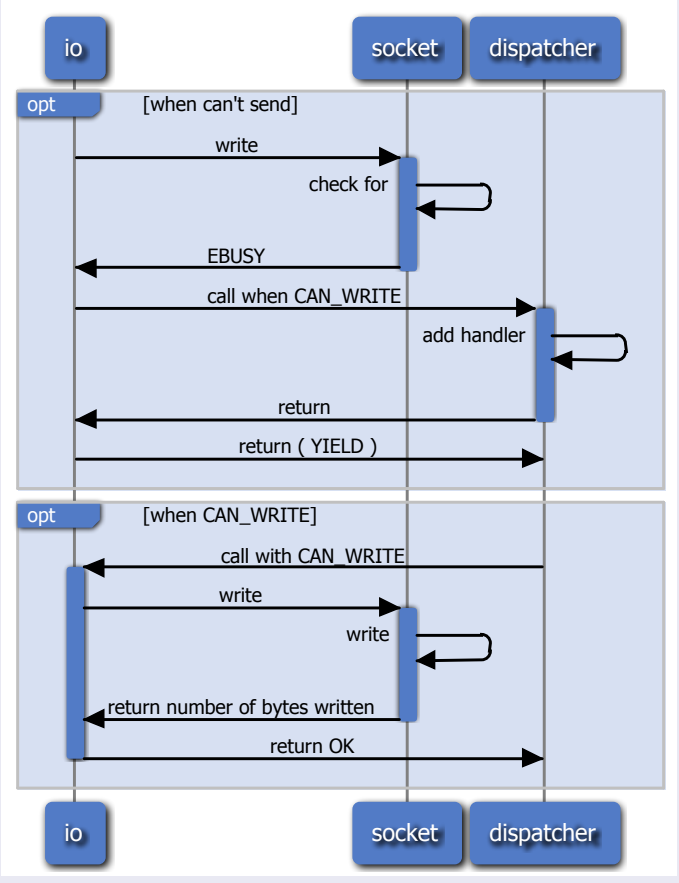
```
    FREE( state->buffer->data );
    FREE( state->buffer );

     if( recvd_msg->header.type != M_PUBACK ) { // wrong message type }
     // take the payload
     // send it or read it
     FREE( recvd_msg );
     END_CORO();
}
```

# MQTT sending with coroutines

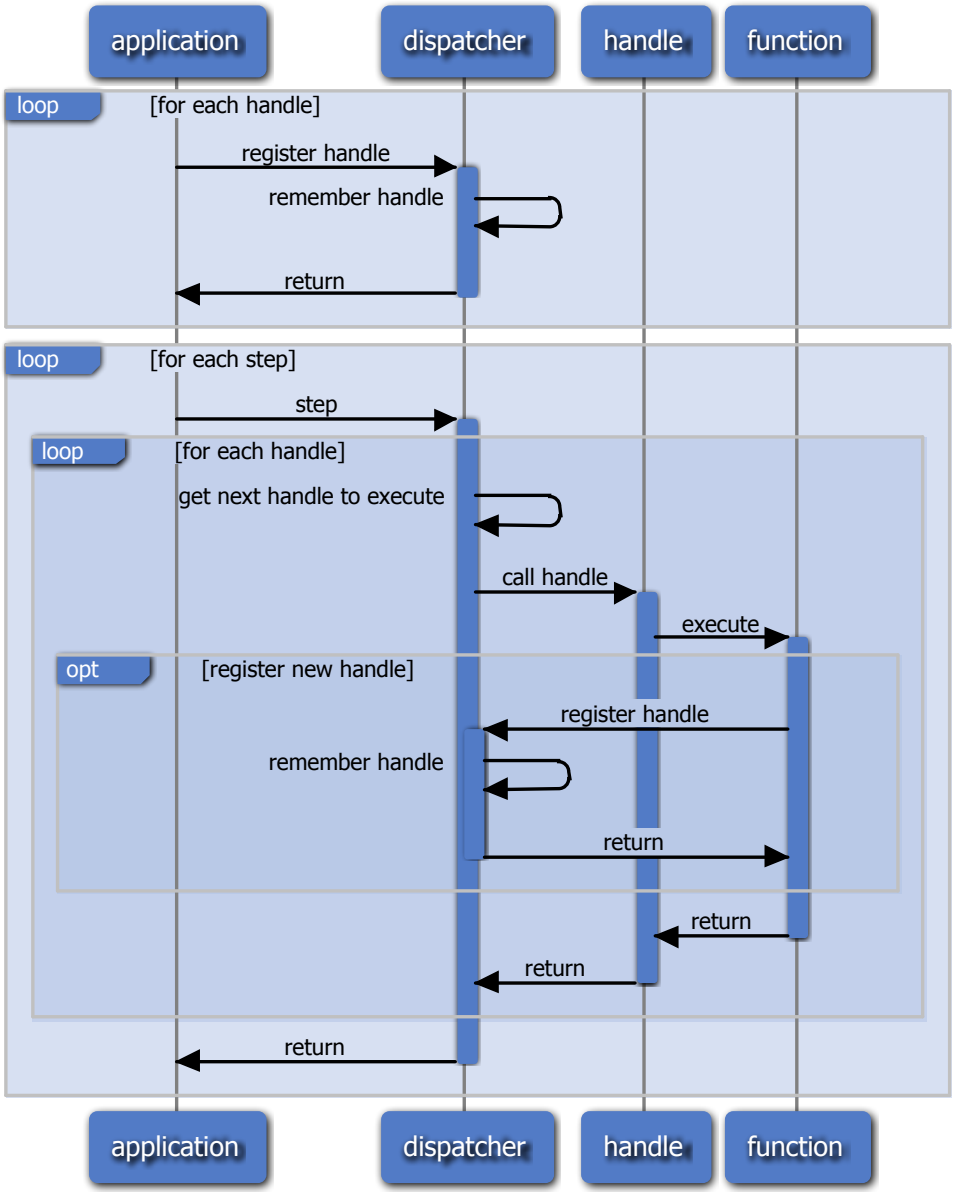
## protocol logic level




## io logic level



# Reactor pattern sequence diagram



- Pros
  - Separates application specific code from the dispatcher implementation
  - Increases code reusablity and allows the code to be modular and well splitted
  - Allows some simple coarse-grain concurrency while not adding the complexity of multiple threads to the system
- Cons
  - The reactor pattern can be more difficult to debug due to the inverted flow of control
  - By only calling request handlers synchronously, the reactor pattern limits maximum concurrency ( can be replaced with proactor that does not have that limitation )

- Pros
  - Separates application specific code from the dispatcher implementation
  - Increases code reusablity and allows the code to be modular and well splitted
  - Allows some simple coarse-grain concurrency while not adding the complexity of multiple threads to the system
- Cons
  - The reactor pattern can be more difficult to debug due to the inverted flow of control
  - By only calling request handlers synchronously, the reactor pattern limits maximum concurrency ( can be replaced with proactor that does not have that limitation )

- Pros
  - Separates application specific code from the dispatcher implementation
  - Increases code reusablity and allows the code to be modular and well splitted
  - Allows some simple coarse-grain concurrency while not adding the complexity of multiple threads to the system
- Cons
  - The reactor pattern can be more difficult to debug due to the inverted flow of control
  - By only calling request handlers synchronously, the reactor pattern limits maximum concurrency ( can be replaced with proactor that does not have that limitation )

- Pros
  - Separates application specific code from the dispatcher implementation
  - Increases code reusablity and allows the code to be modular and well splitted
  - Allows some simple coarse-grain concurrency while not adding the complexity of multiple threads to the system
- Cons
  - The reactor pattern can be more difficult to debug due to the inverted flow of control
  - By only calling request handlers synchronously, the reactor pattern limits maximum concurrency ( can be replaced with proactor that does not have that limitation )

- Pros
  - Separates application specific code from the dispatcher implementation
  - Increases code reusablity and allows the code to be modular and well splitted
  - Allows some simple coarse-grain concurrency while not adding the complexity of multiple threads to the system
- Cons
  - The reactor pattern can be more difficult to debug due to the inverted flow of control
  - By only calling request handlers synchronously, the reactor pattern limits maximum concurrency ( can be replaced with proactor that does not have that limitation )

- Pros
  - Separates application specific code from the dispatcher implementation
  - Increases code reusablity and allows the code to be modular and well splitted
  - Allows some simple coarse-grain concurrency while not adding the complexity of multiple threads to the system
- Cons
  - The reactor pattern can be more difficult to debug due to the inverted flow of control
  - By only calling request handlers synchronously, the reactor pattern limits maximum concurrency ( can be replaced with proactor that does not have that limitation )

- Pros
  - Separates application specific code from the dispatcher implementation
  - Increases code reusablity and allows the code to be modular and well splitted
  - Allows some simple coarse-grain concurrency while not adding the complexity of multiple threads to the system
- Cons
  - The reactor pattern can be more difficult to debug due to the inverted flow of control
  - By only calling request handlers synchronously, the reactor pattern limits maximum concurrency ( can be replaced with proactor that does not have that limitation )

# LibXively dispatcher capabilities

- Execute handle at the desired time
- Execute handle in a desired order
- Execute handle whenever desired event appear on a socket

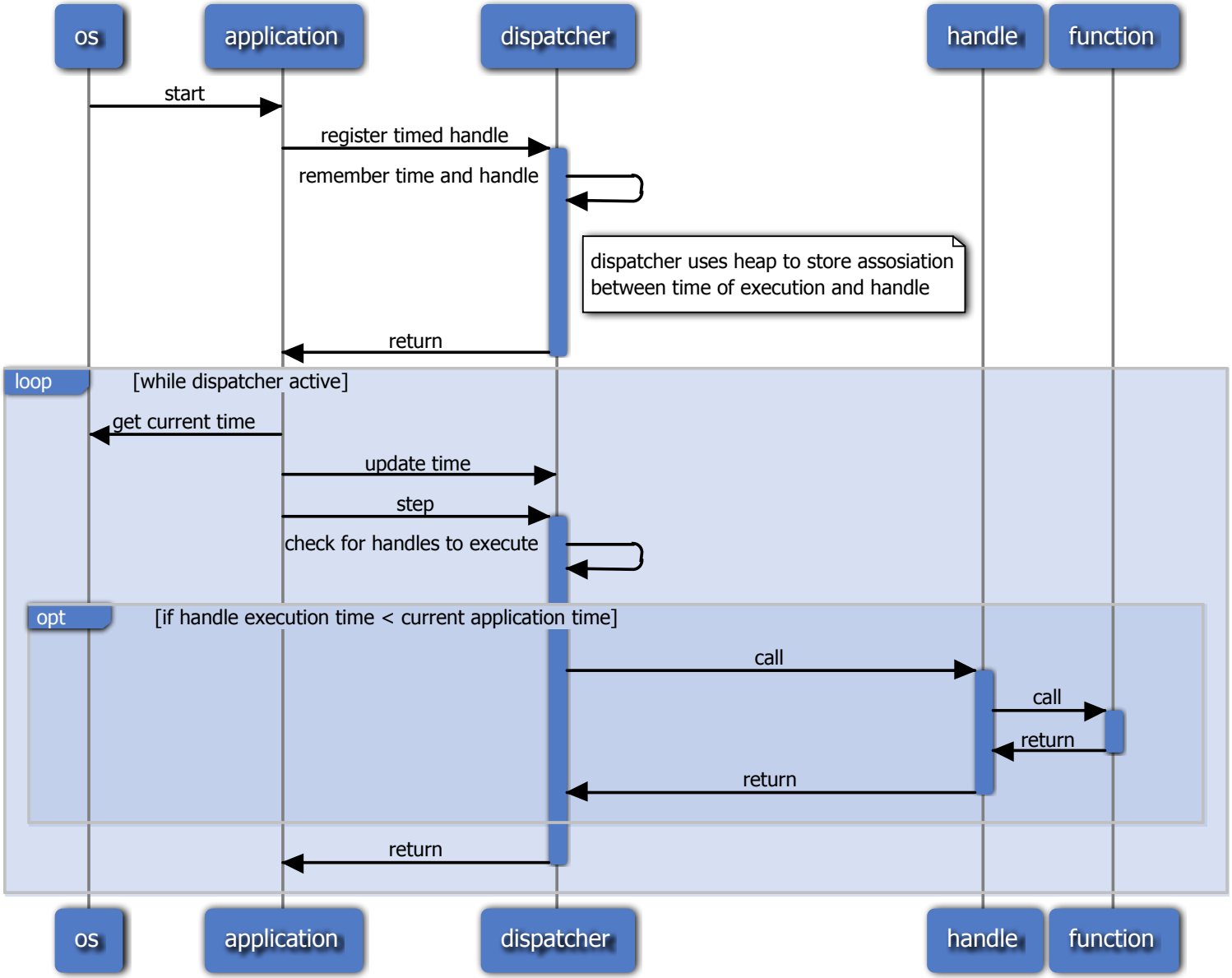
# LibXively dispatcher capabilities

- Execute handle at the desired time
- Execute handle in a desired order
- Execute handle whenever desired event appear on a socket

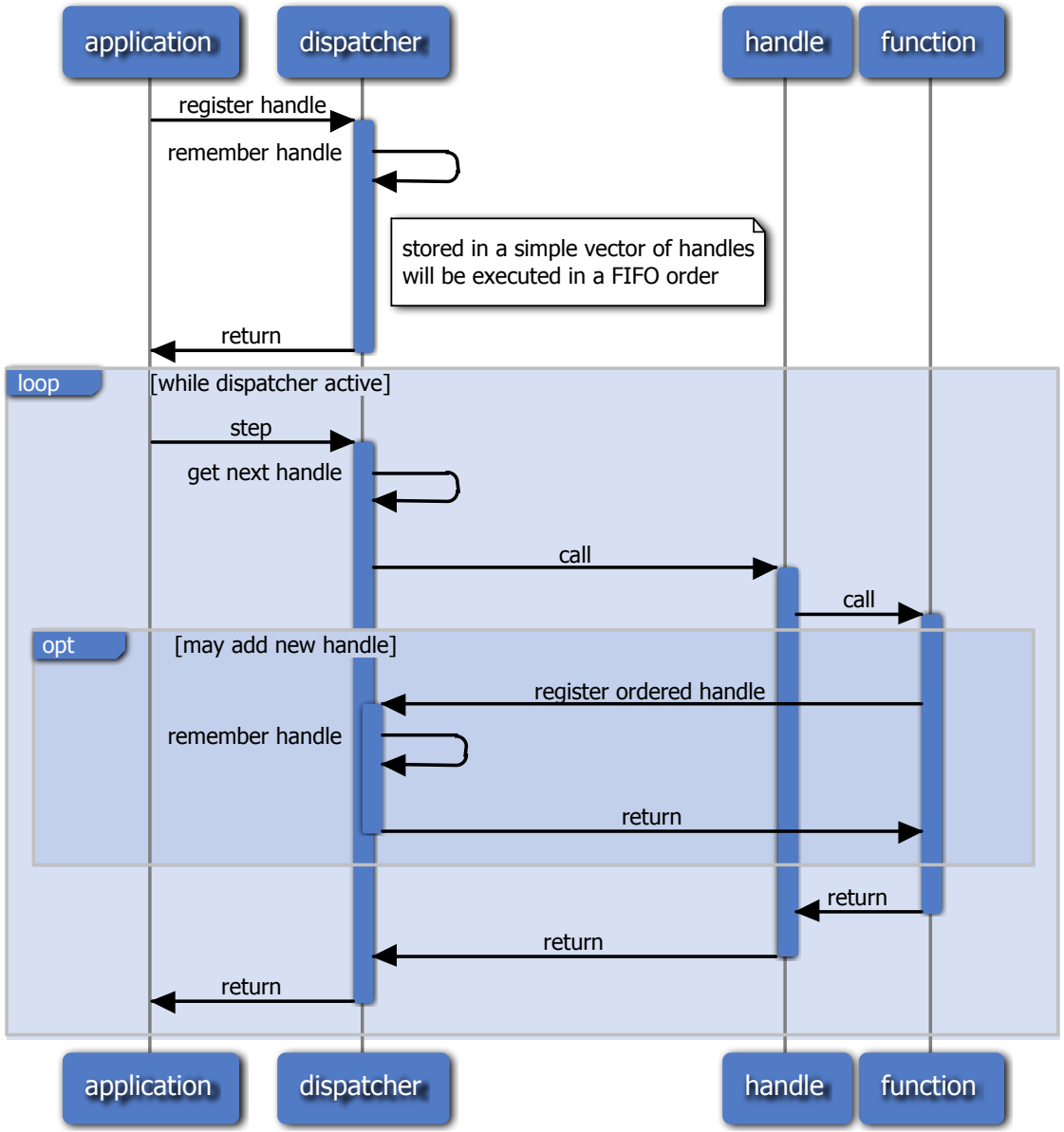
# LibXively dispatcher capabilities

- Execute handle at the desired time
- Execute handle in a desired order
- Execute handle whenever desired event appear on a socket

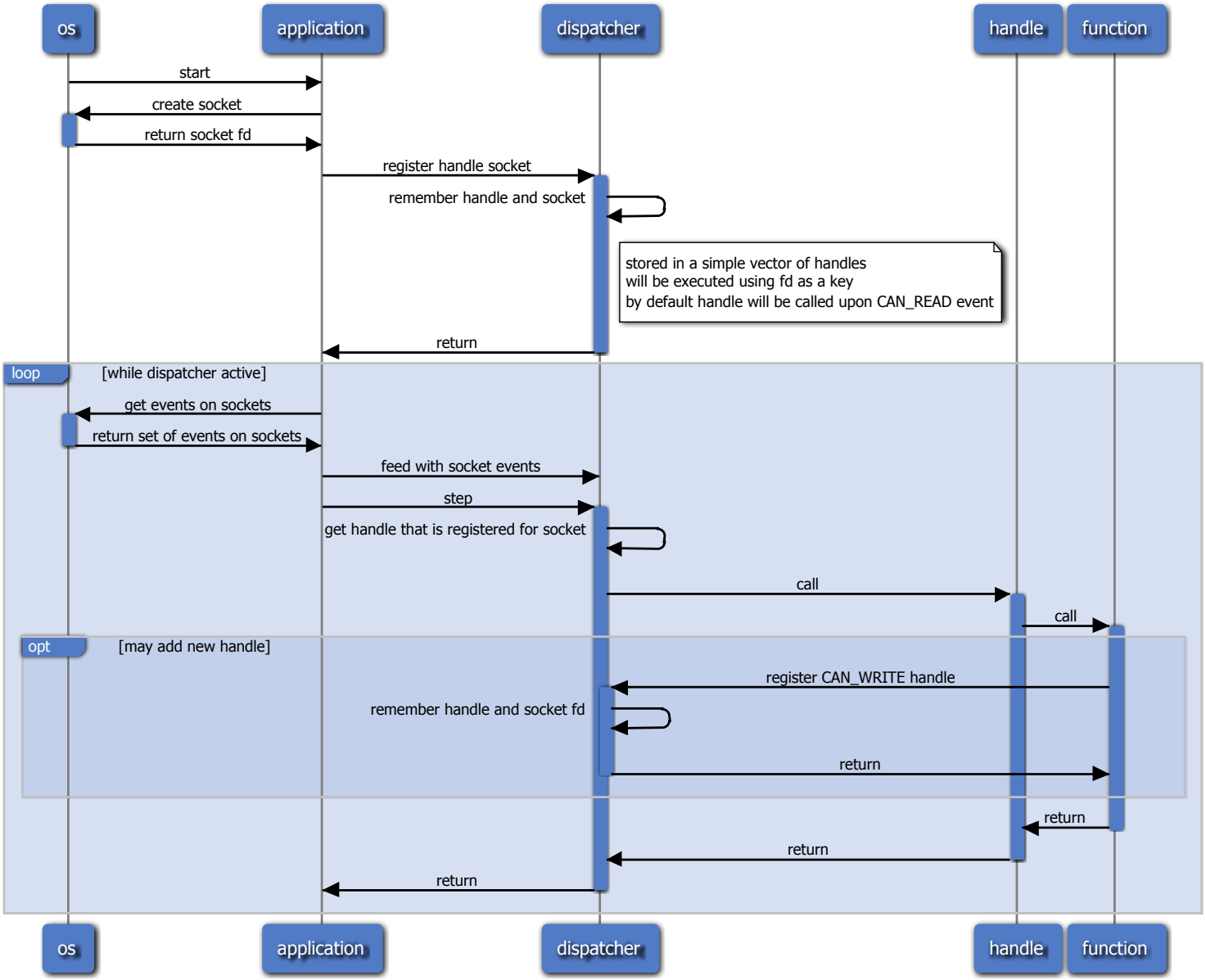
# Timed events



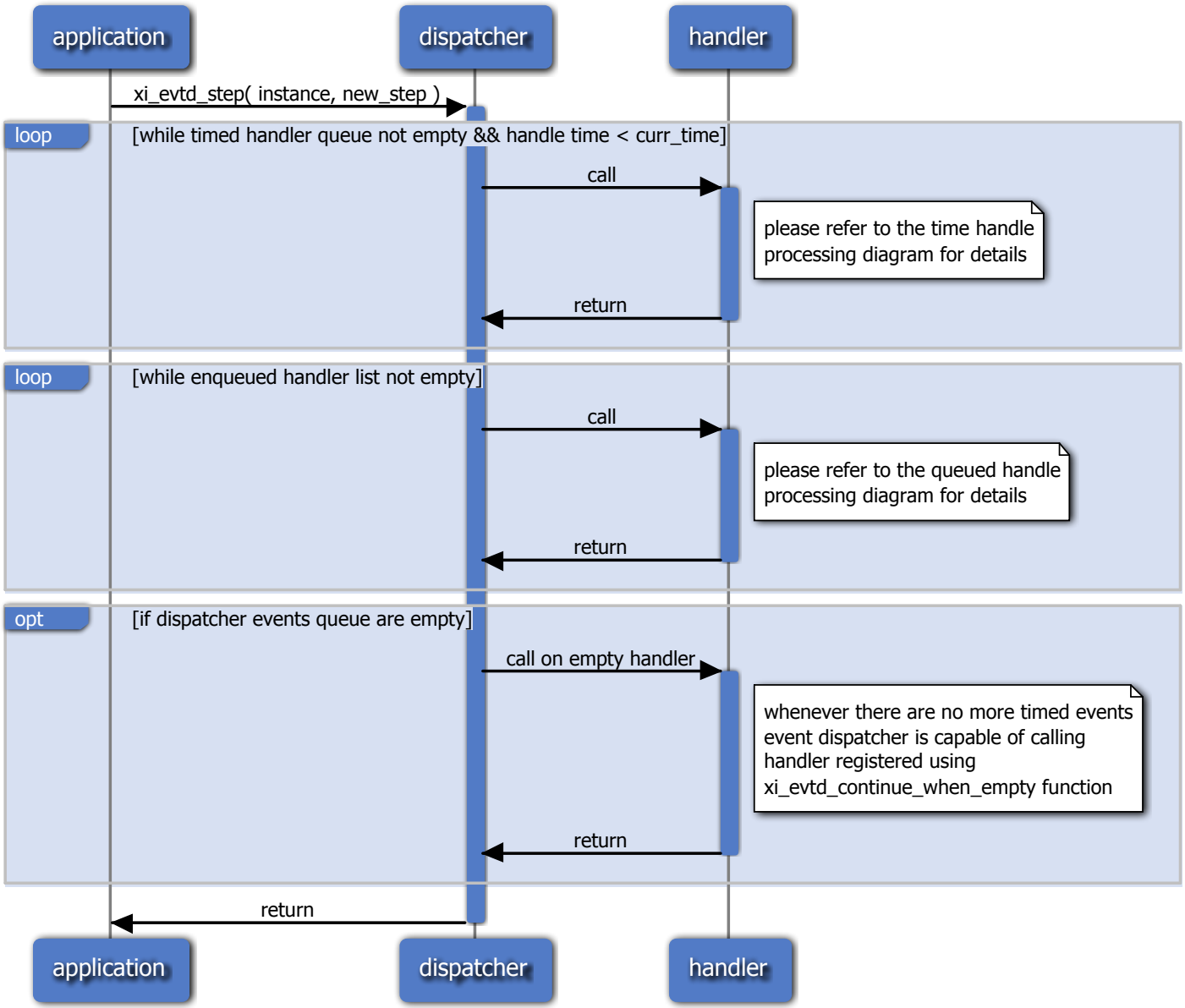
# Queued events



# Socket events



# Processing of event



# Questions?

Questions?