OPEN NETWORKING FOUNDATION

Bundle Extension

Version 0.1

December 23, 2014

Disclaimer

Contents

1 Introduction

This document describes an ONF extension for OpenFlow version 1.3.X that enables modifications to be aggregated into bundles. If all modifications in the bundle succeed, all of the modifications are retained, but if any errors arise, none of the modifications are retained. The bundle concept is similar to the transaction concept used in databases. The term bundle is used instead of the term transaction because the OpenFlow specification uses the term transaction to refer to another concept (e.g. OpenFlow messages contain a transaction ID).

2 Bundle concepts

2.1 Goals

A bundle is a sequence of OpenFlow requests from the controller that is applied as a single OpenFlow operation.

The first goal of bundles is to group related state changes on a switch so that all changes are applied together or that none of them is applied. The second goal is to better synchronise changes across a set

of OpenFlow switches, bundles can be prepared and pre-validated on each switch and applied at the same time.

A bundle is specified as all controllers messages encoded with the same `bundle_id` on a specific controller connection. Messages part of the bundle are encapsulated in a Bundle Add message (see 4.2), the payload of the Bundle Add message is formatted like a regular OpenFlow messages and has the same semantic. The messages part of a bundle are pre-validated as they are stored in the bundle, minimising the risk of errors when the bundle is applied. The applications of the message included in the Bundle Add message is postponed to when the bundle is committed (see 4.4).

A switch is not required to accept arbitrary messages in a bundle, a switch may not accept some message types in bundles, and a switch may not allow all combinations of message types to be bundled together (see 4.2). For example, a switch should not allow to embed a bundle message within a Bundle Add message. At a minimum, a switch must be able to support a bundle of multiple flow-mods and port-mods in any order.

2.2 Example usage

The controller can issue the following sequence of messages to apply a sequence of modifications together.

1. `ONF_BCT_OPEN_REQUEST` `bundle_id`
2. `ONF_ET_BUNDLE_ADD_MESSAGE` `bundle_id` *modification 1*
3. `ONF_ET_BUNDLE_ADD_MESSAGE` `bundle_id` ...
4. `ONF_ET_BUNDLE_ADD_MESSAGE` `bundle_id` *modification n*
5. `ONF_BCT_CLOSE_REQUEST` `bundle_id`
6. `ONF_BCT_COMMIT_REQUEST` `bundle_id`

The switch is expected to behave as follows. When a bundle is opened, modifications are saved into a temporary staging area without taking effect. When the bundle is committed, the changes in the staging area are applied to the state (e.g. tables) used by the switch. If an error occurs in one modification, no change is applied to the state.

2.3 Error processing

The OpenFlow messages part of a bundle must be pre-validated before they are stored in the bundle (see 4.2). For each message sent by the controller, the switch must verify that the syntax of the message is valid and that all features in the message are supported features, and immediately return an error message is this message can not be validated. The switch may optionally verify resource availability and may commit resource at this time and generate errors. Messages generating errors when added to the bundle are not stored in the bundle and the bundle is unmodified.

When the bundle is committed, most errors will have been already detected and reported. One of the message part of the bundle may still fail during commit, for example due to resource availability. In this case, no message part of the bundle is applied and the switch must generate the error message

corresponding to the failure (see 4.4). Messages of a bundle should have unique `xid` to help matching errors to messages. If none of the messages part of the bundle generate an error message, the switch inform the controller of the successful application of the bundle.

2.4 Atomic Modifications

Committing the bundle must be controller atomic, i.e. a controller querying the switch must never see the intermediate state, it must see either the state of the switch with none or with all of the modifications contained in the bundle having been applied. In particular, is a bundle fails, controllers should not receive any notification resulting from the partial application of the bundle.

If the flag `ONF_BF_ORDERED` is specified (see 3.4), the messages part of the bundle must be applied strictly in order, as if separated by `OFPT_BARRIER_REQUEST` messages (however no `OFPT_BARRIER_REPLY` is generated). If this flag is not specified, messages don't need to be applied in order.

If the flag `ONF_BF_ATOMIC` is specified (see 3.4), committing the bundle must also be packet atomic, i.e. a given packet from an input port or packet-out request should either be processed with none or with all of the modifications having been applied. Whether this flag is supported would depend on the switch hardware and software architecture. Packets and messages can temporarily be enqueued while changes are applied. As the resulting increase in forwarding / processing latency may be unacceptable, double buffering techniques are often employed.

If the flag `ONF_BF_ATOMIC` is not specified, committing the bundle does not need to be packet atomic. Packet may be processed by intermediate state resulting from partial application of the bundle, even if the bundle commit ultimately fails. The various OpenFlow counters would also reflect the partial application of the bundle in this case.

2.5 Parallelism

The switch must support exchanging echo request and echo reply messages during the creation and population of the bundle, the switch must reply to an echo request without waiting for the end of the bundle. Echo request and echo reply messages can not be included in a bundle. Similarly, asynchronous messages generated by the switch are not impacted by the bundle, the switch must send status events without waiting for the end of the bundle.

If the switch supports multiple controller channels or auxiliary connections, the switch must maintains a separate bundle staging area for each controller-switch connection. This permits multiple bundles to be incrementally populated in parallel on a given switch. The `bundle_id` namespace is specific to each controller connection, so that controllers don't need to coordinate.

A switch may also optionally support populating multiple bundle in parallel on the same controller connection, by multiplexing bundle messages with different `bundle_id`. The controller may create and send multiple bundles, each identified by a unique `bundle_id`, and then can apply any of them in any order by specifying its `bundle_id` in a commit message. Inversely, a switch may not allow to create another bundle or accept any regular OpenFlow messages between opening a bundle and either committing it or discarding it.

In some implementations, when a switch start storing a bundle, it may lock some of the objects referenced by the bundle. If a message on the same or another controller connection try to modify an object locked by a bundle, the switch must reject that message and return an error. A switch is not required to lock objects referenced by a bundle. If a switch does not lock objects referenced by a bundle, the application of the bundle may fail if those objects have been modified via other controller connections.

3 Bundle messages

3.1 Experimenter ID

The Experimenter ID of this extension is:

`ONF_EXPERIMENTER_ID` = 0x4F4E4600

3.2 Bundle control messages

This extension defines the following messages types:

```
/* Message types */
enum onf_exp_type {
  ONF_ET_BUNDLE_CONTROL      = 2300,
  ONF_ET_BUNDLE_ADD_MESSAGE  = 2301,
};
```

The message `ONF_ET_BUNDLE_CONTROL` is used for all control messages and uses the following structure:

```
/* Message structure for ONF_ET_BUNDLE_CONTROL. */
struct onf_bundle_ctrl_msg {
  struct ofp_header     header;
  uint32_t              experimenter;   /* ONF_EXPERIMENTER_ID. */
  uint32_t              exp_type;       /* ONF_ET_BUNDLE_CONTROL. */

  uint32_t              bundle_id;      /* Identify the bundle. */
  uint16_t              type;           /* ONF_BCT_*. */
  uint16_t              flags;          /* Bitmap of ONF_BF_* flags. */

  /* Bundle Property list. */
  struct onf_bundle_prop_header properties[0]; /* Zero or more properties. */
};
OFP_ASSERT(sizeof(struct onf_bundle_ctrl_msg) == sizeof(struct onf_exp_header) + 8);
```

The `experimenter` field is the Experimenter ID (see 3.1).

The `exp_type` field is set to one of the bundle message types.

The `bundle_id` field is the bundle identifier, a 32 bit number chosen by the controller. The bundle identifier should be unique over the connection during the bundle lifetime.

The `type` field is the control message type. The control types that are currently defined are:


```
/* Bundle control message types */
enum onf_bundle_ctrl_type {
    ONF_BCT_OPEN_REQUEST    = 0,
    ONF_BCT_OPEN_REPLY      = 1,
    ONF_BCT_CLOSE_REQUEST   = 2,
    ONF_BCT_CLOSE_REPLY     = 3,
    ONF_BCT_COMMIT_REQUEST  = 4,
    ONF_BCT_COMMIT_REPLY    = 5,
    ONF_BCT_DISCARD_REQUEST = 6,
    ONF_BCT_DISCARD_REPLY   = 7,
};
```

The `flags` field is a bitmask of bundle flags (see 3.4).

The `properties` field is a list of bundle properties (see **??**).

3.3 Bundle Add message

The message `ONF_ET_BUNDLE_ADD_MESSAGE` uses the following structure:

```
/* Message structure for ONF_ET_BUNDLE_ADD_MESSAGE.
 * Adding a message in a bundle is done with. */
struct onf_bundle_add_msg {
  struct ofp_header     header;
  uint32_t              experimenter;   /* ONF_EXPERIMENTER_ID. */
  uint32_t              exp_type;       /* ONF_ET_BUNDLE_ADD_MESSAGE. */

  uint32_t              bundle_id;      /* Identify the bundle. */
  uint8_t               pad[2];         /* Align to 64 bits. */
  uint16_t              flags;          /* Bitmap of ONF_BF_* flags. */

  struct ofp_header     message;        /* Message added to the bundle. */

    /* If there is one property or more, 'message' is followed by:
     *   - Exactly (message.length + 7)/8*8 - (message.length) (between 0 and 7)
     *     bytes of all-zero bytes */

  /* Bundle Property list. */
  //struct onf_bundle_prop_header properties[0]; /* Zero or more properties. */
};
OFP_ASSERT(sizeof(struct onf_bundle_add_msg) == sizeof(struct onf_exp_header) + 16);
```

The `experimenter` field is the Experimenter ID (see 3.1).

The `exp_type` field is set to one of the bundle message types.

The `bundle_id` field is the bundle identifier, a 32 bit number chosen by the controller. The bundle identifier should be a bundle that has been previously opened and not yet closed.

The `flags` field is a bitmask of bundle flags (see 3.4).

The `message` is a OpenFlow message to be added to the bundle, it can be any OpenFlow message that the switch can support in a bundle. The field `xid` in the message must be identical to the field `xid` of the `ONF_ET_BUNDLE_ADD_MESSAGE` message.

The `properties` field is a list of bundle properties (see **??**).

3.4 Bundle flags

Bundle flags enable to modify the behaviour of a bundle. The bundle flags must be specified on every bundle message part of the bundle, and they need to be consistent.

The bundle flags that are currently defined are:

```
/* Bundle configuration flags. */
enum onf_bundle_flags {
    ONF_BF_ATOMIC  = 1 << 0,  /* Execute atomically. */
    ONF_BF_ORDERED = 1 << 1,  /* Execute in specified order. */
};
```

- `ONF_BF_ATOMIC` is set to request fully atomic application of the bundle.
- `ONF_BF_ORDERED` is set to request the message of the bundle are applied strictly in order.

3.5 Bundle properties

The list of bundle property types that are currently defined are:

```
/* Bundle property types. */
enum onf_bundle_prop_type {
    ONF_ET_BPT_EXPERIMENTER           = 0xFFFF, /* Experimenter property. */
};
```

A property definition contains the property type, length, and any associated data:

```
/* Common header for all Bundle Properties */
struct onf_bundle_prop_header {
    uint16_t         type;    /* One of ONF_ET_BPT_*. */
    uint16_t         length;  /* Length in bytes of this property. */
};
OFP_ASSERT(sizeof(struct onf_bundle_prop_header) == 4);
```

The `ONF_ET_BPT_EXPERIMENTER` property uses the following structure and fields:

```
/* Experimenter bundle property */
struct onf_bundle_prop_experimenter {
    uint16_t         type;    /* ONF_ET_BPT_EXPERIMENTER. */
    uint16_t         length;  /* Length in bytes of this property. */
    uint32_t         experimenter;  /* Experimenter ID which takes the same
                                       form as in struct
                                       ofp_experimenter_header. */
    uint32_t         exp_type;      /* Experimenter defined. */
    /* Followed by:
     *   - Exactly (length - 12) bytes containing the experimenter data, then
```



```
     *   - Exactly (length + 7)/8*8 - (length) (between 0 and 7)
     *     bytes of all-zero bytes */
    uint32_t         experimenter_data[0];
};
OFP_ASSERT(sizeof(struct onf_bundle_prop_experimenter) == 12);
```

The `experimenter` field is the Experimenter ID, which takes the same form as in struct `ofp_experimenter`.

3.6 Bundle errors

Where not otherwise specified below, implementations must use error codes defined in the OpenFlow specification to report issues arising from applying individual modifications.

Errors specific to this extension have the following structure:

```
/* Message structure for all errors. */
struct onf_error_msg {
  struct ofp_header header;
  uint16_t type;            /* OFPET_EXPERIMENTER. */
  uint16_t exp_code;        /* One of ONFERR_ET_* above. */
  uint32_t experimenter;    /* ONF_EXPERIMENTER_ID. */
  uint8_t data[0];          /* Up to 64 bytes of failed request. */
};
OFP_ASSERT(sizeof(struct onf_error_msg) == sizeof(struct ofp_error_experimenter_msg));
```

The `type` field must be set to `OFPET_EXPERIMENTER`.

The `experimenter` field is the Experimenter ID (see 3.1).

The `data` field contains a copy of the failed request message, truncated to 64 bytes.

The `exp_type` field is the experimenter error type. The currently defined experimenter error types are:

```
/* Error codes */
enum onf_error_exp_type {
  ONFERR_ET_UNKNOWN = 2300,       /* Unspecified error. */
  ONFERR_ET_EPERM = 2301,         /* Permissions error. */
  ONFERR_ET_BAD_ID = 2302,        /* Bundle ID doesn't exist. */
  ONFERR_ET_BUNDLE_EXIST = 2303,  /* Bundle ID already exist. */
  ONFERR_ET_BUNDLE_CLOSED = 2304, /* Bundle ID is closed. */
  ONFERR_ET_OUT_OF_BUNDLES = 2305, /* Too many bundles IDs. */
  ONFERR_ET_BAD_TYPE = 2306,  /* Unsupported or unknown message control type. */
  ONFERR_ET_BAD_FLAGS = 2307, /* Unsupported, unknown, or inconsistent flags. */
  ONFERR_ET_MSG_BAD_LEN = 2308,  /* Length problem in included message. */
  ONFERR_ET_MSG_BAD_XID = 2309,  /* Inconsistent or duplicate XID. */
  ONFERR_ET_MSG_UNSUP = 2310,    /* Unsupported message in this bundle. */
  ONFERR_ET_MSG_CONFLICT = 2311, /* Unsupported message combination in this bundle. */
  ONFERR_ET_MSG_TOO_MANY = 2312, /* Cant handle this many messages in bundle. */
  ONFERR_ET_MSG_FAILED = 2313,   /* One message in bundle failed. */
  ONFERR_ET_TIMEOUT = 2314,      /* Bundle is taking too long. */
  ONFERR_ET_BUNDLE_IN_PROGRESS = 2315, /* Bundle is locking the resource. */
};
```


4 Bundle operations

4.1 Creating and opening a bundle

To create a bundle, the controller sends a `ONF_ET_BUNDLE_CONTROL` message with type `ONF_BCT_OPEN_REQUEST`. The switch must create a new bundle with id `bundle_id` attached to the current connection, with the options specified in the flags and properties. If the operation is successful, `ONF_ET_BUNDLE_CONTROL` message with type `ONF_BCT_OPEN_REPLY` must be returned by the switch. If an error arises, an error message is returned.

If the `bundle_id` already refers to an existing bundle attached to the same connection, the switch must refuse to open the new bundle and send an `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_BAD_ID` code. The existing bundle identified by `bundle_id` must be discarded.

If the switch can not open this bundle because its having too many opened bundles on the switch or attached to the current connection, the switch must refuse to open the new bundle and send an `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_OUT_OF_BUNDLES` code. If the switch can not open the bundle because the connection is using an unreliable transport, the switch must refuse to open the new bundle and send an `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_OUT_OF_BUNDLES` code.

If the `flags` field request some feature that can not be implemented by the switch, the switch must refuse to open the new bundle and send an `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_BAD_FLAGS` code.

4.2 Adding messages to a bundle

The switch adds message to a bundle using the `ONF_ET_BUNDLE_ADD_MESSAGE`. After the bundle is opened, the controller can sends a sequence of `ONF_ET_BUNDLE_ADD_MESSAGE` messages to populate the bundle. Each `ONF_ET_BUNDLE_ADD_MESSAGE` includes an OpenFlow message, this OpenFlow message is validated, and if successful it is stored in the bundle specified by `bundle_id` on the current connection. If a message validation error or a bundle error condition arise, an error message is returned.

Message validation includes at minimum syntax checking and that all features are supported, and it may optionally include checking resource availability (see 2.3). If a message fails validation, an error message must be returned. The error message must use the `xid` of the offending message, the error data field corresponding to that message and the error code corresponding to the validation failure.

If the `bundle_id` refers to a bundle that does not exist on the current connection, the corresponding bundle is created using arguments from the `ONF_ET_BUNDLE_ADD_MESSAGE` message. If an error arise from creating the bundle, the relevant error message is returned (see 4.1). No `ONF_ET_BUNDLE_CONTROL` message with type `ONF_BCT_OPEN_REPLY` is returned by the switch in this case.

If the `bundle_id` refers to a bundle that is already closed, the switch must refuse to add the message to the bundle, discard the bundle and send an `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_BUNDLE_CLOSED` code.

If the flags field is different from the flags that were specified when the bundle was opened, the switch must refuse to add the message to the bundle, discard the bundle and send an ofp_error_msg with OFPET_EXPERIMENTER type and ONFERR_ET_BAD_FLAGS code.

If the message in the request is normally supported on the switch but is not supported in a bundle, the switch must refuse to add the message to the bundle and send an ofp_error_msg with OFPET_EXPERIMENTER type and ONFERR_ET_MSG_UNSUP code. This is the case for hello, echo and bundle messages, messages that are not requests, or if the implementation does not support including a specific modification message in a bundle.

If the message in the request is incompatible with another message already stored in the bundle, the switch must refuse to add the message to the bundle and send an ofp_error_msg with OFPET_EXPERIMENTER type and ONFERR_ET_MSG_CONFLICT code.

If the bundle is full and can not fit the message in the request, the switch must refuse to add the message to the bundle and send an ofp_error_msg with OFPET_EXPERIMENTER type and ONFERR_ET_MSG_TOO_MANY code.

If the message in the request does not have a valid length field, the switch must refuse to add the message to the bundle and send an ofp_error_msg with OFPET_EXPERIMENTER type and ONFERR_ET_MSG_BAD_LEN code.

Message added in a bundle should have a unique xid to help matching errors to messages, and the xid of the bundle add message must be the same. A switch may optionally verify that the two xid of a message are consistent or that two messages of the bundle don't have the same xid, and if this is the case refuse to add the new message to the bundle and send an ofp_error_msg with OFPET_EXPERIMENTER type and ONFERR_ET_MSG_BAD_XID code.

4.3 Closing a bundle

To finish recording a bundle, the controller may sends a ONF_ET_BUNDLE_CONTROL message with type ONF_BCT_CLOSE_REQUEST. The switch must close the bundle specified by bundle_id on the current connection. After closing the bundle, it can no longer be modified and no messages can be added to it, it can only be committed or discarded. Closing a bundle is optional. If the operation is successful, ONF_ET_BUNDLE_CONTROL message with type ONF_BCT_CLOSE_REPLY must be returned by the switch. If an error arises, an error message is returned.

If the bundle_id refers to a bundle that does not exist, the switch must reject the request and send an ofp_error_msg with OFPET_EXPERIMENTER type and ONFERR_ET_BAD_ID code.

If the bundle_id refers to a bundle that is already closed, the switch must refuse to close to the bundle, discard the bundle and send an ofp_error_msg with OFPET_EXPERIMENTER type and ONFERR_ET_BUNDLE_CLOSED code.

If the flags field is different from the flags that were specified when the bundle was opened, the switch must refuse to close to the bundle, discard the bundle and send an ofp_error_msg with OFPET_EXPERIMENTER type and ONFERR_ET_BAD_FLAGS code.

The switch may optionally do additional validation of the messages part of the bundle as the result of the close request, such as validating resource availability. For each message that fails this additional

validation, an error message must be generated that refer to the offending message. After sending those individual error messages, the switch must discard the bundle and send an additional `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_MSG_FAILED` code.

4.4 Committing Bundles

To finish and apply the bundle, the controller sends a `ONF_ET_BUNDLE_CONTROL` message with type `ONF_BCT_COMMIT_REQUEST`. The switch must commit the bundle specified by `bundle_id` on the current connection, it must apply all messages stored in the bundle as a single operation or return an error. The commit operation and the way the message part of the bundle are applied depend on the bundle flags (see 3.4). If the bundle was not closed prior to this request, it is automatically closed (see 4.3).

The commit is successful only if all messages parts of the bundle can be applied without error. If the bundle does not contain any message, commit is always successful. If the commit is successful, the switch must apply all messages of the bundle as a single operation, and a `ONF_ET_BUNDLE_CONTROL` message with type `ONF_BCT_CLOSE_REPLY` must be returned by the switch. The `ONF_ET_BUNDLE_CONTROL` message with type `ONF_BCT_CLOSE_REPLY` must be sent to the controller after the processing of all messages part of the bundle are guaranteed to no longer fail or produce an error.

If one or more message part of the bundle can not be applied without error, for example due to resource availability, the commit fails and all messages part of the bundle must be discarded without being applied. When the commit fails, the switch must generate an error message corresponding to the message that could not be applied. The error message must use the `xid` of the offending message, the error data field corresponding to that message and the error code corresponding to the failure. If multiple messages are generating errors, the switch may return only the first error found or generate multiple error messages for the same bundle. After sending those individual error messages, the switch must send an additional `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_MSG_FAILED` code.

After a commit operation, the bundle is discarded, whether the commit was successful or not. After receiving a successful reply or error message for this operation, the controller can reuse the `bundle_id`.

Modification requests may require replies to be returned to the controller or events to be generated. Because any message of the bundle may fail, replies and events can only be generated once all modifications in the bundle have been applied. For replies, each of these replies contains the transaction ID of the corresponding request.

If the `bundle_id` refers to a bundle that does not exist, the switch must reject the request and send an `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_BAD_ID` code.

If the `flags` field is different from the flags that were specified when the bundle was opened, the switch must refuse to commit the bundle, discard the bundle and send an `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_BAD_FLAGS` code.

4.5 Discarding Bundles

To finish and discard the bundle, the controller sends a `ONF_ET_BUNDLE_CONTROL` message with type `ONF_BCT_DISCARD_REQUEST`. The switch must discard the bundle specified by `bundle_id` on the current connection, and all messages part of the bundle are discarded. If the operation is successful, a `ONF_ET_BUNDLE_CONTROL` message with type `ONF_BCT_DISCARD_REPLY` must be returned by the switch. If an error arises, an error message is returned. After receiving either a successful reply or an error message, the controller can reuse the `bundle_id`.

All implementations must be able to process a discard request on a existing bundle on the current connection without triggering errors. If the `bundle_id` refers to a bundle that does not exist, the switch must reject the request and send an `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_BAD_ID` code.

4.6 Other bundle error conditions

If a `ONF_ET_BUNDLE_CONTROL` message contains an invalid `type`, the switch must reject the request and send an `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_BAD_TYPE` code.

If a `ONF_ET_BUNDLE_CONTROL` or `ONF_ET_BUNDLE_ADD_MESSAGE` message specifies `flags` different for those already specified on an existing bundle, the switch must reject the request and send an `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_BAD_FLAGS` code.

If the switch does not receive any `ONF_ET_BUNDLE_CONTROL` or `ONF_ET_BUNDLE_ADD_MESSAGE` message for an opened `bundle_id` for a switch defined time greater than 1s, it may send an `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_TIMEOUT` code. If the switch does not receive any new message in a bundle apart from echo request and replies for a switch defined time greater than 1s, it may send an `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_TIMEOUT` code.

If an OpenFlow message can not be processed because a bundle is locking a resource this message is using, the switch must reject that message and send an `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_BUNDLE_IN_PROGRESS` code. If a switch can not process other messages between opening a bundle and either committing it or discarding it, the switch must reject that message and send an `ofp_error_msg` with `OFPET_EXPERIMENTER` type and `ONFERR_ET_BUNDLE_IN_PROGRESS` code.