Olso net Communications

Object Exchange Protocol

Version 0.12
October 14, 2008

Proprietary and confidential

Introduction

This document describes the reliable Object Exchange Protocol (OEP) for wireless devices in immediate proximity (no multi-hopping). The way OEP is implemented makes it possible to invoke it temporarily from a praxis that normally uses a completely different (possibly multi-hop – as in TARP) mode of communication. The protocol has been simplified with respect to its prototype version (implemented in the LOADER praxis for the Seawolf project – see the document in Apps/SEAWOLS/LOADER). The simplification consists in leaving the initial handshake up to the praxis. While this imposes on the praxis the burden of actually negotiating the handshake, it also makes the protocol more flexible, as the “objects” to be exchanged can be now pretty much anything (their description does not have to fit the rather restrictive format of the original version). Besides, in order to invoke OEP, the praxis would have to establish some kind of “agreement” first, anyway, so perhaps there is really no additional complication at the praxis's end.

An object to be exchanged between the involved pair of nodes must be divisible into so-called *chunks*, which are fixed-size (54-byte) portions of the object. Chunks are transmitted and received independently, possibly out of sequence. This assumption boils down to the following formal postulates:

- The number of chunks (denoted **nc**) comprising the object must be determined by the praxis during the initial handshake (i.e., before the exchange takes place). This number cannot be bigger than 64K (it is stored in a **word**).

- The sending party must be able to provide (upon request) any legitimate chunk number **k**, where **k** is any number between **0** and **nc**–1. In other words, the chunks must be accessible at random in any order. The protocol may legitimately ask for the same chunk more than once.

- The receiving party must be able to accept a chunk at random, given its number, i.e., the praxis must be able to accommodate chunks arriving in any order. The protocol guarantees that once accepted the same chunk **will not** be submitted to the praxis again.

Once the exchange is started, the protocol will operate “behind the scenes”. The praxis can await its completion in the same way it awaits a regular event. While running, the protocol will be invoking a function (the so-called chunk handler) provided by the praxis to handle outgoing chunk requests (at the sender) or incoming chunks (at the recipient).

Praxis interface

To make sure that OEP is compiled into the praxis, put

```
#include “oep.h”
```

into the header. This file, as well as `oep.c` containing the actual code of the protocol can be found in `PicOS/Libs/Lib`.

OEP defines its private plugin which should be configured **after** any plugins used by the praxis for “normal” communication. This function:

```
Boolean oep_init ();
```

should be called by the praxis after initializing its PHYs and “regular” plugins. Among other things, it registers the OEP plugin into the first available slot. As plugins are scanned by VNETI from high to low, the OEP plugin will be given priority to claim packets. Normally, when the protocol is dormant, the plugin simply lets all received packets through, i.e., they can be freely claimed by other plugins..

The function returns YES, if it succeeds, and NO, if it fails. Practically the only reason for a failure (unless you do something obviously wrong, like calling the function twice) is the lack of a free slot for the OEP plugin.

The protocol assumes that the two parties have established beforehand that they want to exchange an object. One of the two nodes is the sender, the other is the receiver. These are the parameters that the two parties must have agreed upon:

- The so-called link ID of the receiver. This is a 16-bit (word) value that should (momentarily) uniquely identify the receiver in the neighborhood.
- The request number. This is an 8-bit (byte) value intended to uniquely identify the session, in case the two parties have been involved in previous exchanges. Typically, the receiver presents this number, which it increments by 1 modulo 256, before getting involved in a new exchange. This way, the pair: [link ID, request number] can be guaranteed to be temporarily unique.

Having agreed on the exchange, the two parties invoke these functions:

Sender:

```
byte oep_snd (word lid, byte rqn, word nc, oep_chf_t sfun);
```

Receiver:

```
byte oep_rcv (word nc, oep_chf_t rfun);
```

where **lid** and **rqn** are the negotiated link ID and request number. Note that these parameters are not provided to **oep_rcv**. As said above, the receiver is supposed to maintain those parameters internally and pass them to the sender during the praxis-level handshake. OEP provides hooks for keeping them internally, such that they do not have to be invented each time the receiver is activated. The praxis can access them via these functions (macros):

```
word oep_getlid ();
void oep_setlid (word lid);
byte oep_getrqn ();
void oep_setrqn (byte rqn);
```

whose meaning should be obvious. In particular, it may be appropriate to initialize the node's link ID at the beginning (following **oep_init**) and not change it from then on. Then, during the handshake, the receiver simply tells the sender its link ID and the current value of request number. At the beginning of a new exchange, the receiver may want to increment the request number, e.g.,

```
oep_setrqn (oep_getrqn () + 1)
```

which automatically takes care of the modulo truncation.

Parameter **nc**, common to both startup functions, tells the number of chunks in the exchanged object. The last argument is a function with this header:

```
void fun (word state, address buffer, word ck);
```

which will be used as the chunk handler. For `oep_snd`, the function is expected to fill the `buffer` (whose size is always exactly 54 bytes) with the chunk number `ck` (between **0** and `nc`–1, inclusively). For `oep_rcv`, the function should accept the chunk number `ck` (whose contents are provided in the `buffer`). If any of the two handlers can block, then `state` should be used as the retry state, i.e., the handler should execute `release` for as long as the operation cannot progress, with a retry at `state`.

The values returned by `oep_snd` and `oep_rcv` indicate whether the respective invocation was successful. They can be one of these:

0 (aka `OEP_STATUS_OK`)	the operation has been started
`OEP_STATUS_NOPROC`	unable to fork a thread
`OEP_STATUS_NOMEM`	unable to allocate memory
`OEP_STATUS_NOINIT`	OEP has not been initialized
`OEP_STATUS_SND`	previous send operation still in progress
`OEP_STATUS_RCV`	previous receive operation still in progress

Any nonzero value means failure, i.e., nothing has been started. Once the operation has been started, the praxis can use this function:

```
byte oep_wait (word state);
```

to wait for its completion. The function returns when the protocol has terminated; otherwise, it will keep bouncing to the indicated state. The returned value can be:

`OEP_STATUS_DONE`	the exchange has completed successfully
`OEP_STATUS_TMOUT`	there was a timeout, exchange abandoned

Note that, in some circumstances, the sender may terminate with `OEP_STATUS_TMOUT` even though the receiver has in fact completely received the object. This will happen when the sender does not get the final empty **GO** packet indicating a complete reception by the other party (see below).

The status of the OEP module can be determined at any time by invoking this macro:

```
byte oep_status ();
```

which returns one of the values: ...`DONE`, ...`TMOUT`, ...`SND`, ...`RCV`, ...`NOINIT`, depending on the operation status. With this function (macro):

```
Boolean oep_busy ();
```

you can check whether the module is currently in the middle of an exchange (`oep_status` would return ...`SND` or ...`RCV`). For as long as this holds, the OEP plugin is active and intercepts all received packets, dropping those that it cannot make sense of. Note: if required, I can make it pass those packets to other plugins. Perhaps this should be an option, as generally, the node can run into memory problems, if they are not promptly extracted from the queue. When the transaction is complete (`oep_busy` returns NO, `oep_wait` unblocks), the OEP plugin becomes inactive.

It is possible to use the OEP module with multiple PHYs. By default, it operates with PHY 0, but you can use this function (macro):

```
void oep_setphy (int phy);
```

to change the default. The argument must be a legitimate PHY previously configured by the praxis. You can learn the current PHY setting for OEP by invoking

```
int oep_getphy ();
```

This shouldn't be done while an exchange is in progress. (I believe nothing wrong will happen: the new PHY will become valid for the next exchange, but why push your luck?)

Note that the present implementation does not change the PHY's parameters (channel, rate) for the duration of the exchange (should it?). You can safely do it from the praxis when you see that the startup function (**`ope_snd`** or **`ope_rcv`**) has succeeded, and revert to the previous setting after the transaction is over (when **`oep_wait`** returns).

Note, however, that, upon a successful startup, OEP does replace the PHY's network ID with the promiscuous transparent value 0xFFFF. The previous value of the network ID is saved and automatically reverted to when the exchange is over.

In addition to the status, OEP makes available to the praxis another byte value, which stores the last operation type carried out by the protocol. This value can be retrieved by invoking the macro:

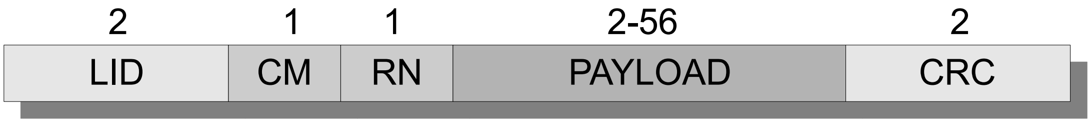
```
byte oep_lastop ();
```

which returns **`OEP_STATUS_SND`** or **`OEP_STATUS_RCV`**. Exceptionally, if there has been no transaction yet, the returned value is zero. The intended role of **`oep_lastop`** is to provide a quick way to fork the processing after returning from **`oep_wait`** (which can be centralized, i.e., put at a single state shared by both the send and receive parts of the praxis code). For example, some transfers may require cleanups (deallocation of some data structures needed by the chunk handlers) and those cleanups may differ depending on the direction.

Protocol details

Packet format

Figure 1 shows the layout of packets exchanged in the protocol. All multi-byte components are sent in *little endian* order. The trailing CRC field is not directly visible to the program and is only included for completeness.



2	1	1	2-56	2
LID	CM	RN	PAYLOAD	CRC

Figure 1: Packet format

The CM field differentiates between two packet types used in the protocol: **GO** (0x11) and **CHUNK** (0x12).[1] The first packet type is issued by the recipient to request a series of chunks (that are still missing), the second type is issued by the sender (and it carries one chunk).

[1] Note that these numbers have changed with respect to the original version of the protocol in the LOADER praxis.

All packets belonging to the same exchange contain the same values of LID (link ID) and RN (request number) previously negotiated by the nodes. Any packet whose first three bytes do not match those agreed-upon values is ignored.

Rounds

The actual transmission of the object is carried out in rounds, whereby the receiver first notifies the sender which chunks are still missing, and then the sender transmits those chunks back-to-back (with 2 msec packet separation), without expecting anything from the receiver, until all requested chunks have been sent. At the end of a round the receiver tallies up the chunks and, if some are still missing, requests all or some of them from the sender, which then transmits another batch of chunks, and so on. This operation continues until all chunks have been received or any of the two parties hits a timeout while expecting a reply from the other.

For a **GO** packet (sent by the receiver), the payload consists of the list of missing chunks numbered from 0 up to the total number of chunks in the object (**nc**) – 1. This list may contain individual chunk numbers (little-endian words) or ranges. For as long as the numbers in the list are increasing, they stand for individual chunks. A decreasing pair of numbers represents a continuous range of chunks between and including the two. For example, the sequence 3, 5, 6, 15, 13, 8, 17 describes chunks 3, 5, 6, 15, 8 through 13, and 17. Typically, the initial **GO** packet sent by the receiver consists of two words containing **nc** – 1 followed by 0, i.e., requesting the complete range of all chunks (its payload length is 4 bytes).[2]

A round is always started by a single **GO** packet, which means that the sender may only be able to learn about a subset of all those chunks that are still missing at the receiver. This will happen if the complete description of all the missing chunks cannot fit into a maximum-sized **GO** packet (23 16-bit numbers). Note, however, that the first **GO** packet (starting the first round) tells the sender to transmit all chunks of the object. Thus, in the first round, the provider will always send the entire object; thus, if the nodes are lucky, no further rounds may be needed.

Upon startup (after a successful call to `ope_rcv`), the receiver will transmit the first **GO** packet 8 times at 1 sec intervals expecting a chunk from the sender. If no chunk packet arrives within that time, the receiver will abort the exchange signaling timeout (**`OEP_STATUS_TMOUT`**).

As soon as the receiver finds out that it has received all the chunks, it should notify the sender that the reception is over by sending it an empty **GO** packet (with the payload of zero bytes). There will be no reply to such a packet: the sender will just shut down its part of the protocol. To improve the chance that the sender has seen the empty **GO** packet, the receiver transmits it three times at 40 msec intervals.

Having seen the first **CHUNK** packet, the receiver immediately stops sending the last **GO** packet and starts receiving the chunks. OEP keeps its own tally of received chunks (a bit map), which it uses to construct subsequent **GO** packets. The payload of a **CHUNK** packet starts with a word containing the chunk number followed by exactly 54 bytes of the chunk. If the tail of the object does not fall on a chunk boundary, the receiver is expected to know where to truncate the last chunk. Typically, in such cases, the initial negotiation within the praxis will also cover the actual length of the object in bytes (communicated by the sender to the receiver).

When the sender is done with all the chunks requested in the last **GO** packet, it will send an empty chunk, i.e., a **CHUNK** packet with empty payload. Having received such a packet, the receiver will know that there is nothing more to expect in the present round. To make sure

[2] If there is only one chunk to transmit, the payload contains a single word 0.

that the end of round is properly conveyed to the receiver, the sender will transmit the empty chunk 8 times at 1 sec intervals until it receives a **GO** packet. If no such packet arrives within that time, the sender will conclude that the exchange has been broken (signaling **OEP_STATUS_TMOUT**).

The timeouts and/or retransmission counts probably have to be tuned or made settable by the praxis.

For an illustration how to use OEP, please have a look at a simple praxis to load/dump the node's EEPROM via UART. The praxis (directory **Apps/SEAWOLF/EEDUMPER**) uses a library module (files **oep_ee.h**/**oep_ee.c** in **PicOS/Libs/Lib**) implementing EEPROM block transfers over OEP. A similar module for handling Seawolf images (pictures) is described in a separate document.