HLD of Patchless TCP Zero Copy Socklnd

Author Liang Zhen

2006/11/20

1 Introduction

Current socklnd can work with TCP ZC(zero copy) in Linux kernel, but a kernel patch is needed to make it work, and customer don't like it. In order to get rid of the patch, socklnd has to send out an explicit ACK for paged bulk, with ACK the zero-copy patch can be removed.

2 Requirements

This project includes:

1) Send and handle explicit ACK for ZC bulk without hurting performance.

2) Backwards compatible with current socklnd.

3 Functional specification

3.1 Explicit ACK for paged bulk

The kernel patch used now is implemented to support receiving notification when zero-copy network I/O has completed, and tell the original caller that the pages may now be overwritten. To remove the patch:

1. Sender needs to keep the packet if it's sending out by zero copy(tcp_sendpage).

2. Receiver needs to send a ACK to sender after receiving all pages in the bulk. Sending of ACK should be eager and with low overhead.

3. Sender needs to release the matching packet after getting the ACK.

3.2 Backward compatibility

We still want the new socklnd can work with current socklnd, so we have to make the new one can understand and handle current protocol of socklnd.

1. Read version of peer's socklnd and handle handshake correctly while connection is established, set flag for the connection to mark version of connection.

2. Send / retrieve correct message header and payload according to the version of connection.

4 Logic specification

4.1 Explicit ACK

4.1.1 Message header

```
typedef struct {
    __u32 ksm_type; /* type of socklnd message */
    __u32 ksm_cksum; /* checksum if != 0 */
    __u64 ksm_req_cookie; /* ack required if != 0 */
    __u64 ksm_ack_cookie; /* ack if != 0 */
    union {
    ksock_lnet_msg_t lnetmsg; /* normal lnet message header */
    } ksm_u;
    ......
    } ksock_msg_t
    #define KSOCK_MSG_NOOP 0xc0 /* ksm_u empty, just ACK message
*/
    #define KSOCK_MSG_LNET 0xc1 /* message with LNet payload */
```

- No explicit message type for zero copy, sender needs to set unique ksm_req_cookie for outgoing ZC message.
- Receiver can identify a ZC message by checking ksm_req_cookie of the incoming message, ACK needs to be sent by receiver if a ZC message is received.
- ZC-ACK can be in a empty message (KSOCK_MSG_NOOP, only be handled at socklnd level, LNET will never know about it), or piggybacked on a normal lnet message by just setting value for msg->ksm_ack_cookie (the value should be same with ksm_req_cookie of received ZC message).
- If sender gets a message with valued ksm_ack_cookie, sender needs to search through the packets list on the peer, and release the packet if out->ksm_req_cookie and in->ksm_ack_cookie are matching.

4.1.2 Enqueue ZC/ACK message

- Enqueue ZC message
  - ksocknal_queue_tx_locked() will generate ksm_req_cookie for ZC message. We have no idea about if the message will be sent by ZC or not before the connection is ready(Not all connection support ZC), so we can't set ksm_req_cookie while packing the message(ksocknal_send()). Because type of connection is known while enqueue the message into the connection, so we set ksm_req_cookie in ksocknal_queue_tx_locked().
  - ksocknal_queue_tx_locked() will put ZC message in a list on peer.

```
ksocknal_queue_tx_locked(tx, conn)
{
......
tx->tx_conn = conn;
if (ksocknal_lib_zc_capable(tx))
{
tx->tx_msg.ksm_req_cookie = next_cookie;
list_add_tail(&tx->tx_zc_list, &conn->ksnc_zc_req_list);
}
......
}
```

- Enqueue ACK message
  - Receiver always try to piggyback ACK on a normal LNET message in the outgoing queue of the connection, to reduce overhead of ACK.

```
int ksocknal_piggyback_zcack_locked(__u64 cookie, ksock_conn_t *conn)
{
if (conn->ksnc_tx_piggyback != NULL) {
conn->ksnc_tx_piggyback->tx_msg.ksm_ack_cookie = cookie;
conn->ksnc_tx_piggyback = next_unpiggybacked_tx(conn);
return 1;
}
return 0;
}
```

  - If there is no LNET message in the outing tx_queue of the connection, or all packets have been piggybacked with ACK, allocate a NOOP packet and try to enqueue it by calling of ksocknal_queue_tx_locked()
  - When ksocknal_queue_tx_locked() gets a NOOP message, it will check if there is any LNET message without setting of ksm_ack_cookie in the outgoing queue of connection (check it again because we released the lock while allocating NOOP), if found, the ACK cookie will be piggybacked on the LNET message, the NOOP message will be recycled.
  - When ksocknal_queue_tx_locked() gets a normal LNET message, it'll check if there is any NOOP message at the end of tx_queue. If there is, take out the NOOP from tx_queue and assign the cookie to the LNET message which is going to be enqueued, and recycle the NOOP message.

- NOOP message with ACK will be sent out only if there is no message in the connection or all messages queued in the connection have taken ACK already.
- ACK messages are always put on the fastest connection of the peer because ACK must be sent eagerly.

4.1.3 ZC message handle

ZC message will be handled in ksocknal_process_receive(). After receiving payload of a message, it will call ksocknal_handle_zc_req() if ksm_req_cookie is set in the message header.

Handle function:

```
int ksocknal_handle_zc_req(ksock_peer_t *peer, __u64 cookie)
{
......
conn = ksocknal_find_conn_locked(NOOP_TX_SIZE, peer);
rc = ksocknal_piggyback_zcack_locked(cookie, conn);
if (rc > 0)
return 0
...... /* allocate a NOOP packet */
ksocknal_queue_tx_locked(tx, conn);
......
}
```

4.1.4 ACK message handle

ACK message will be taken by ksocknal_process_receive(). After receiving the message header, it will call ksocknal_handle_zc_ack() if ksm_ack_cookie is set in the message header.

Handle function (ksocknal_handle_zc_ack()):

- Search in ZC message list on peer, pick out and free the message if matching cookie is found, or close the connection if no matching cookie(protocol problem).

4.2 Backward compatibility

Let's call current version of ksocklnd protocol as V1.x, and new version as V2.x

4.2.1 Hello message

Both peers of the connection exchange hello messages while connecting. Format of hello message in V2.x is like this:

```
typedef struct {
__u32 kshm_magic; /* magic number of socklnd message */
__u32 kshm_version; /* version of socklnd message */
lnet_nid_t kshm_src_nid; /* sender's nid */
```

lnet_nid_t kshm_dst_nid; /* destination nid */
lnet_pid_t kshm_src_pid; /* sender's pid */
lnet_pid_t kshm_dst_pid; /* destination pid */
__u64 kshm_src_incarnation; /* sender's incarnation */
__u64 kshm_dst_incarnation; /* destination's incarnation */
__u32 kshm_ctype; /* connection type */
__u32 kshm_nips; /* # IP addrs */
__u32 kshm_ips[0]; /* IP addrs */
} WIRE_ATTR ksock_hello_msg_t;

However, if V2.x peer got a lnet_magicversion_t request with different version number, it should switch to correct handshake procedure if it's an compatible version(v1.x). V2.x peer always sends out ksock_hello_msg_t request in the first trying of connecting with a new peer, if the handshake failed, then try to use compatible version request(v1.x).

4.2.2 Protocol function table

Talk to peer with different protocol version, we need different function table to handshake and pack/unpack message header.

typedef struct {
int (*pro_send_hello)(ksock_conn_t *, ksock_msg_t *); /* handshake function */
int (*pro_recv_hello)(ksock_conn_t *, ksock_msg_t *); /* handshake function */
void (*pro_pack)(ksock_conn_t *, ksock_tx_t *); /* message pack */
void (*pro_unpack)(ksock_conn_t *, ksock_msg_t *); /* message unpack */
} ksocknal_protocol_t;
ksocknal_protocol_t ksocknal_protocol_v1 =
{
ksocknal_send_hello_v1,
ksocknal_recv_hello_v1, ‘
ksocknal_pack_msg_v1,
ksocknal_unpack_msg_v1 };
ksocknal_protocol_t ksocknal_protocol_v2 =
{
ksocknal_send_hello_v2,
ksocknal_recv_hello_v2, ‘
ksocknal_pack_msg_v2,
ksocknal_unpack_msg_v2
};

4.2.3 Active / Passive connect handshake

- If active connecting peer and passive connecting peer are both V2.x, just go ahead.

- Passive: While V2.x peer accepting connecting-request from V1.x peer, it can't send back any error to V1.x peer, because if V1.x peer gets error, it aborts connecting request. So V2.x should call correct functions to process V1.x request.

```
ksocknal_recv_hello(.....)
    {
    .....
    rc = libcfs_sock_read(sock, &hmv->magic + 1, sizeof(*hmv) - sizeof(hmv-
>magic), timeout);
    proto = ksocknal_match_protocol(hmv->version_major, hmv->version_minor);
    if (!active && conn->ksnc_proto != proto) /* Correct my protocol */
    conn->ksnc_proto = proto;
    conn->ksnc_proto->pro_recv_hello(conn, ...)
    .....
    }
    ksocknal_send_hello(.....)
    {
    .....
    conn->ksnc_proto->pro_send_hello(conn, ...);
    .....
    }
    ksocknal_create_conn(.....)
    {
    .....
    if (active) {
    conn->ksnc_proto = ksocknal_match_protocol(version_major, version_minor);
    ksocknal_send_hello(....)
    } else
    conn->ksnc_proto = ksocknam_match_protocol(DEFAULT_MAJOR, DE-
FAULT_MINOR);
    ksocknal_recv_hello(.....) /* recv_hello will return correct version number
*/
    .....
    ksocknal_send_hello(.....)
    .....
    }
```

- Active: While V2.x peer is the active connecting peer, it always send out V2.x message header(ksock_hello_msg_t) at the first time, if passive connecting peer is V1.0 and it can't understand V2.x request, it aborts the connecting handshake. If it's happened, V2.x peer needs to reconnect to the other peer by compatible(V1.x) protocol.

```
ksocknal_connect(....)
    {
```

```
.....
{
lnet_connect(...);
rc = ksocknal_create_conn(..., version)
if ((rc == -EPROTO || rc == -ECONNRESET) &&
peer->ksnp_version_major == KSOCK_PROTO_V2_MAJOR) {
peer->ksnp_version_major = KSOCK_PROTO_V1_MAJOR;
peer->ksnp_version_minor = KSOCK_PROTO_V2_MINOR;
rc = EPROTO;
} else if (rc < 0) {
lnet_connect_console_error(...);
goto failed;
}
.....
}
.....
}
```

4.2.4 Connecting Race

Both version share same code to handle connecting race:

- A) V2.x peer connect to V2.x peer, same logic with old version
- B) V2.x peer use V1.x protocol to connect to V1.x peer, same logic with old version
- C) V2.x peer use V2.x protocol to connect to V1.x peer, at the same time V1.x peer connect to V2.x peer. In this case, no matter who will win with "favour of higher NID", V2.x can't connect to V1.x successfully, because V1.x always refused to connecting request from V2.x peer, V2.x peer has to re-connect to V1.x with V1.x protocol. At this point, the connecting race is exactly same as B), and can be fully handled.

4.2.5 Send / Receive message

As we know, outgoing message is packed in ksocknal_send(), at that time we don't know about peer's version, so we always initialize message in V2.x (ksock_msg_t), while the message is enqueued to connection's outgoing tx_queue, it might be re-packed if version of connection is V1.x. Overhead of re-pack is very low so we can ignore it (only re-assign value of a few fields).

```
typedef struct {
lnet_hdr_t kscm_hdr; /* lnet header */
} WIRE_ATTR ksock_compat_msg_t;
typedef struct {
.....
union {
```

```
ksock_lnet_msg_t normal;
ksock_compat_msg_t compat;
} ksm_u;
}
void ksocknal_pack_msg_v1(ksock_conn_t *conn, ksock_tx_t *tx)
{
tx->tx_iov[0].iov_base = (void *)&tx->tx_lnetmsg->msg_hdr;
tx->tx_iov[0].iov_len = sizeof(lnet_hdr_t);
tx->tx_resid = tx->tx_nob = tx->tx_lnetmsg->msg_len + sizeof(lnet_hdr_t);
}
ksocknal_queue_tx_locked(...)
{
......
ksocknal_protocol[version].pro_pack(conn, tx)
......
}
```

We always know type of connection while receiving a message, so it's possible to read & unpack the message according the the peer version.

```
static void ksocknal_unpack_msg_v1(ksock_conn_t *conn, ksock_msg_t
*msg) {
/* Just a few things need to be done here */
msg->ksm_type = KSOCK_MSG_LNET;
msg->ksm_req_cookie = msg->ksm_ack_cookie = 0;
.....
}
ksocknal_process_receive (ksock_conn_t *conn)
{
......
switch (conn->ksnc_rx_state) {
......
case SOCKNAL_RX_LNET_HEADER:
ksocknal_protocol[conn->ksnc_version].pro_unpack(conn, &conn->ksnc_msg);
......
}
```

5 Use cases

Only use cases needs to be concerned is connecting race.

- A) V2.x peer connects to V2.x peer, same logic with current race handling.
- B) V2.x peer use V1.x protocol to connect to V1.x peer, same logic with current race handling.

- C) V2.x peer use V2.x protocol to connect to V1.x peer, at the same time V1.x peer connect to V2.x peer. In this case, no matter who will win the race with "favour of higher NID", V2.x can't connect to V1.x, because V1.x always refused to connecting request from V2.x peer, V2.x peer has to re-connect to V1.x peer with V1.x protocol. At this point, the connecting race is exactly same as B), and can be fully handled. Here is the description with more details:

  P2 connects to P1 by V2.x, P1 connects to P2 by V1.x

  1. P1 win the race, so P1 proceeds to connect to P2 by V1.x, and P2 is rejected because unknown protocol.
     (a) P2 is re-scheduled and retry with V1.x protocol, P2 finds connection is established, so give up.
     (b) P2 is re-scheduled and retry with V1.x protocol, P2 finds connecting is still in progress but losts race, so give up (Just like handle race with same protocol version V1.x).
  2. P1 lost the race, so P1 will be re-scheduled. P2 is rejected cause unknow protocol, so P2 is re-scheduled too.
     (a) P2 is re-scheduled and retry with V1.x protocol, P1 is re-scheduled and retry with V1.x protocol too, still use same race logic (with protocol V1.x).

6 State machine

6.1 RX state

There are five RX states for current connection to receive message:
#define SOCKNAL_RX_HEADER 1 /* reading header */
#define SOCKNAL_RX_PARSE 2 /* Calling lnet_parse() */
#define SOCKNAL_RX_PARSE_WAIT 3 /* waiting to be told to read the body */
#define SOCKNAL_RX_BODY 4 /* reading body (to deliver here) */
#define SOCKNAL_RX_SLOP 5 /* skipping body */
There could be six RX states for new version connection to receive message:
#define SOCKNAL_RX_KSM_HEADER 1 /* reading ksock message header */ \
#define SOCKNAL_RX_LNET_HEADER 2 /* reading lnet message header */
#define SOCKNAL_RX_PARSE 3 /* Calling lnet_parse() */
#define SOCKNAL_RX_PARSE_WAIT 4 /* waiting to be told to read the body */
#define SOCKNAL_RX_LNET_PAYLOAD 5 /* reading lnet payload (to deliver here) */

#define SOCKNAL_RX_SLOP 6 /* skipping body */

SOCKNAL_RX_KSM_HEADER is a new state, it's the start state for processing all incoming messages, also, it's the only state to receive a NOOP message(zero copy ACK), which will be totally handled in LND level. .

V2.x SOCKNAL_RX_LNET_HEADER matches V1.x SOCKNAL_RX_HEADER.

V2.x SOCKNAL_RX_LNET_PAYLOAD matches V1.x SOCKNAL_RX_BODY. ZC message is handled in this state(After receiving of all payload, send out ZC ACK)

If connection's peer is V1.x, the connection will never be in SOCKNAL_RX_KSM_HEADER, state for receiving a packet starts from SOCKNAL_RX_LNET_HEADER.