High Level Design for Tdinal

Version 3.0 / Feb. 23, 2005

February 11, 2008

1 Requirements

The tdinal is a new nal module for Lustre windows project. It provides the low-level networking operations to Lustre portals module.

Windows kernel does not have kernel socket interfaces as linux. But instead, windows kernel network transport drivers (also referred as protocol drivers) exprot a set of transport driver interfaces (TDI). The TDI defines the kernel-mode network interface which is exposed to TDI clients, such as the upper edge of transport protocol stacks (Socket Emulator / NetBios Emulator ...).

The tdinal is to designed to be a TDI client, using the TDI functionality to implement the networking operations. And it's to be started from the ksocknal module of Lustre linux project. And surely some adaptions are to be done for the tdinal module.

There are several issues that we need pay attention to:

1.1 Interacting with TDI

All the linux kernel sockets related part will be removed from tdinal. And it needs directly call tdi related routines without any extra layers of socket emulator and buffer conversions.

1.2 Asynchronous mechanism.

In ksocknal, transmisson and receiving are processed asynchrony. Or the normal working of ksocknal might be blocked, becuse several connections share the same scheduler thread. It starts s one scheduler thread per processor. The scheduler thread will be used the same in tdinal.

There is another way if we don't implement the asynchronous transfer: using one thread per connection. But it will bring too much burden and limit the system dimension.

1.3 Coding/naming conventions

It's better to keep an identical coding/naming convention to ksocknal sytle. The codes related to nal part should follow this rule.

Bur for some pure windows tdi routines, we still keep them windows programming sytle (to save time and keep the windows conventions). All these parts are to be put in separate files.

2 Tdinal components

Tdinal architecture:

Note: Missing a picture here;
The routines can be divided into 4 functionality sets:

- tdi client object management
- tdi connection management
- data process unit
- ksocknal inheritance

2.1 Tdi client object management

TDI has 3 types of the network communication entities:

- Transport addresses
- Connection endpoints
- Transport provider control channels

We need create and use the corresponding tdi objects of these entities to realize our necessary network operations.

2.1.1 File objects of the underlying transport provider:

We need first obtain the fileobject of the transport provider we concern. Then with it we can create address objects and connection objects ...

Windows system has several transport providers, such as:

- NetBT.sys provides Netbios protocal over tcp/ip
- Tcpip.sys provides tcp/udp/ip/rawip transports

At the moment what we concern is the stream (tcp) transport provider: "\Device\Tcp"

2.1.2 File objects of transport address:

For tcp transport, the address type is TDI_ADDRESS_IP. The transport address can contain several typed addresses: such as tcp ip address, netbios name or ipx address. For our's aspect, we only care the ip address.
We can try to open the transport provider's device name: "\Device\Tcp" to get it's fileobject. Let's move further to the way getting the address and connection objects.

Flow of creating address object:

Note: Missing a picture here.
Normally we just call ZwCreateFile to open the corresponding transport device name with a parameter of pre-defined EA (extended attributes) buffer :

```
typedef struct _FILE_FULL_EA_INFORMATION {
    ULONG NextEntryOffset;
    UCHAR Flags;
    UCHAR EaNameLength;
    USHORT EaValueLength;
    CHAR EaName[1];
} FILE_FULL_EA_INFORMATION, *PFILE_FULL_EA_INFORMATION;
```

The EaName member must be TdiTransportAddress (System defines it as a string: "TransportAddress"). Then system will return the address fileobject if it succeeds.

2.1.3 File objects of connection endpoints:

This type of file object represents a connection between the local and the peer. Here's the flow of creating connection object:

Note: missing a picture here.
After both address object and connection object are created, we can call TdiBuildAssociatedAddress to conjunct the two object, then we can start the connection or listen operations on the address object.

2.2 Tdi connection management

We'll describe how the tdi connections are created in this section. There are two cases: daemon mode and client mode.

2.2.1 Daemon mode:

In ksocknal, the daemon module (acceptor) is being executed in user mode. And it could map the user mode socket to kernel space. The connection is accepted and built in user mode, then mapped into ksocknal. Linux kernel provides such a mechansim.

But in windows system there's no such a mechanism of mapping a user-mode socket to kernel tdi object. Windows socket apis are realized in user mode and the relationship between user mode socket and kernel mode tdi object is undocumented. So there's no formal way to get the corresponding kernel tdi address / connection objects for a user mode socket handle. As a result, we have to put the acceptor part into kernel space, i.e., we'll start a daemon thread to listening on the user specified port.

For a listening tdi object, we must set the following callbacks to correctly accept new incoming requests:

Event callback	Description
ClientEventConnect	To be triggered when new requests comming

Flow of connection creation (server):

1. acceptor issues an ioctl of NAL_CMD_START_DAEMON (new ioctl code in tdinal).
2. ksocknal_cmd gets triggerred to process the ioctl cmd.
3. ksockna_cmd will call ksocknal_start_daemon to create a daemon ksock_conn_t and start the dademon thread, then let the daemon listen on the user specified port.
4. the daemon thread will create a new tdi address object and prepare (create/associate) the backlog listeing children conenctions, then set the ClientEventConnect event callback to be ready for incoming connection requests .
5. in case a peer issues a connection request to the listening port, the ClientEventConnect callback will be triggerred to accept the connecting request. Then the accepted connection is built ready and to be put into the accepted queue. In the end it will wake up the deamon thread.
6. the daemon thread is woken up and gets the conenction. Then ksocknal_create_conn will be called to craete/initialize the ksocknal_conn_t strucutre, create the peer and scheule the data request.
7. now the connection is ready for data transferring.

2.2.2 Client mode:

For a sender client, it just need to issue a TDI_CONNECT Irp to build the connection to the listening daemon.

Flow of connection creation (client):

1. ptlctl will issue an ioctl of NAL_CMD_CONNECT_PEER (new ioctl cmd) to kernel
2. ksocknal_cmd gets triggerred to process NAL_CMD_CONNECT_PEER
3. ksocknal_cmd just calls the corresponding tdi routines to create address / conneciton objects and associate them, then issue the TDI_CONNECT Irp to the transport driver. The underlying transport will make the real connecton between the two nodes. Then it will call ksocknal_creaet_conn.
4. ksocknal_create_conn in turn will build the ksocknal connection and create the peer/route/conection structures.
5. after that's done, the connection is ready for data transferring.

2.3 Data process unit

2.3.1 Data receiving

For data receiving part, TDI provides several event calbacks to process the incoming data:

Callback	Description
ReceiveEventHandler	Normally small buffer transfer
ChainedReceiveHandler	Bulk buffer transfer (via MDL)
ReceiveExpeditedHandler	Out-of-band: normal small buffer transfer
ChainedReceiveExpeditedHandler	Out-of-band: bulk buffer transfer (via MDL)

We need implement all these event callbacks to achieve a completely asynchronous process. These callbacks are to be set up correctly with the TDI address object. Then the callbacks will be triggered once the peer sends data via the connection. All the callbacks are running at DISPATCH_LEVEL and we are expected to return as quickly as possible. There's an issue that at the time callback is triggerred the portals might not prepare well the buffers yet. We must maintain a buffer queue to receive and store all the tsdus (transport service data unit, it could be a data buffer or chained MDL).

Flow of data receiving:

1. when the socknal connetion is built and initialized, the call of ksocknal_new_packet will schedule a receive request of portals message header in size_of(ptl_hdr_t) bytes.
2. when the remote peer sends data, the tdi callbacks TDI_EVENT_RECEIVE / TDI_EVENT_RECEIVE_EXPEDITED / TDI_EVENT_CHAINED_RECEIVE will be triggerred. Which callback will be triggerred depends on the size of the tsdu data and the options. For a bulked data traffic, the

bulk transfer callback TDI_EVENT_CHAINED_RECEIVE / ChainedReceiveHandler will be actived.

3. then the tdi callback will receive all the incoming data in the tsdu and queue it into the our own tsdu buffer queue. It also needs to wake up the scheduler thread.
4. the scheduler thread now is woken up and processes the receive request of ptl_hdr_t via ksocknal_process_receive
5. ksocknal_process_receive will call ksocknal_receive to read payload into the pre-allocated buffer.
6. similarly, ksocknal_receive calls ksocknal_recv_iov or ksocknal_recv_kiov. And ksocknal_recv_iov or ksocknal_recv_kiov will move the data from tsdu buffers to the the connection data buffer.
7. after the data is received, we are back to ksocknal_process_receive. ksocknal_process_receive is to process the SOCKNAL_RX_HEADER case and call lib_parse with the received ptl_hdr_t.
8. then potoals lib_parse routine will prepare memory buffer and issue a new receive request to read all the payload into the prepared buffer. lib_parse calls ksocknal_recv or ksocknal_recv_pages to schedule the new receive request.
9. then it starts a new loop of these steps, but the state for ksocknal_process_receive is changed to SOCKNAL_RX_BODY now.

2.3.2 Data transmission

For data transmission part, though TDI is well designed for asynchonous, the transport driver of tcp/ip could not support buffer sending, i.e., the event callback mechanism of ClientEventSendPossible is not supported. However we can use the asynchrous feature of windows I/O mechanism, because windows I/O Irp process is designed as a asynchronous procedure. We can use it to realize NON_BLOCKING transmissions.

In case that the underlying transprot driver could not send the buffer immediately, it will queue the buffer internally and return STATUS_PENDING to the caller leaving h the irp request un-completed. At later time that the data transmission is completed, the transport driver will complete the original Irp request via IoCompleteRequest. Then our irp completion routine will be triggered and we can re-gain the chance to finalize the transmission and wake up the scheduler thread.

But there's a potential race between the tdi sending routine and the completion routine. Because whether the TDI_SEND request succeeds or fails, the completion routine will be called always. And in completion routine we have no way to identify the original TDI_SEND call is successful or not. Normally for a successful call of IoCallDriver to issue the Irp to the underlying transport driver, CompletionRoutine will be called before IoCallDriver returns; for the case the call is pended, IoCallDriver returns STATUS_PENDING before CompletionRoutine is triggered. But it's not always true.

Here we can intorduce a reference count in Irp compeltion context. First we add two references on it. Then both the Irp completion routine and TDI_SEND call (after IoCallDriver returns) dereference the reference count. The last one to dereference is to do the Irp cleanup job.

Flow of data transmission:

1. portals will call the ksocknal callbacks: ksocknal_send or ksocknal_send_pages to process a sending request.
2. then ksocknal_send or ksocknal_send_pages will call ksocknal_sendmsg.
3. ksocknal_sendmsg will allocate a ltx structure to encapsule all the information inside, then call ksocknal_launch_packet.
4. ksocknal_launch_packet will queue the ltx to the corresponding connection, then wake up the corresponding scheduler thread (ksocknal_queue_tx_locked is to do this job).
5. the scheduler thread will be woken up and call ksocknal_process_transmit.
6. Then ksocknal_transmit will be called by ksocknal_process_transmit.
7. ksocknal_transmit will call ksocknal_send_iov or ksocknal_send_kiov to process the transmission request.
8. ksocknal_send_iov or ksocknal_send_kiov will collect all the buffers and lock them into MDL to them paged-in, then issue the TDI_SEND Irp to the transport driver.
9. If it returns STATUS_PENDING, (-EAGAIN) will be returned to ksocknal_scheduler. Then the tx will be re-queued to the conn->ksnc_tx_queue, but conn->ksnc_tx_ready will be kept as 0, so the conn structure won't be queued to sched->kss_tx_conns. It's the duty of the Irp completion routine to re-active the scheduler to process the conn's requests.
10. If step 9 succeeds with STATUS_SUCCESS, that means the payload is fullly sent out, now if the caller does not drops the last reference of the completion context, it will do the same to step 9. Otherwise, it will return with (rc $>$ 0). Then ksocknal_transmit will loop until all the payload is sent out, and in trun return to ksocknal_process_transmit. ksocknal_process_transmit will finialize the tx and return to ksocknal_scheduler with (rc $>$ 0).

11. When the Irp completion routine is triggered, if it drops the last reference of the completion context, it will do the cleanup jobs and re-active the connection object to be scheduled. If not, it just do nothing and return to system. The corresponing job will be done by the caller function.

2.4 ksocknal inheritance

Most of this part will be kept the same to ksocknal part, at least the logics are the same. Just some modifications about linux socket interfaces / buffer management (iov/kiov) / connections management are to be made to fully use the advantage of tdi.

The details of this section are to be convered in the DLD document.

3 Summary

N/A