TDI Overview

This document provides an overview of the Transport Driver Interface (TDI), TDI drivers, TDI operational model, TDI requests and events and interactions between TDI components for client and server side sockets. The examples are based on the usage of the TDI interface between AFD.sys and TCPIP.sys, two of most heavily used TDI drivers in the system. This document primarily applies to pre-Vista versions of Windows i.e. Windows 2000, Windows XP and Windows 2003. Although TDI continues to be supported on Vista and later versions of Windows for backward compatibility, it is on the path to deprecation. So developers are urged to use the new interfaces like WSK and WFP instead of TDI.

Transport Driver Interface

Transport Driver Interface (TDI) is a low level kernel mode networking stack interface to access the transport layer functionality in Windows. Transport drivers, like TCPIP.sys, exposed the TDI interface at their upper edge. TDI provides standard methods for protocol addressing, sending and receiving datagrams, writing and reading streams, initiating connections, detecting disconnects and has been supported by Windows since its inception. In Windows, kernel mode access to the TCPIP stack requires programming to TDI APIs, making TDI the the only socket interface in the kernel. TDI is supported by the export driver TDI.sys that primarily provides helper functions for use by TDI drivers.

TDI Drivers fall in the following 3 categories, as shown in figure 1:

- TDI Clients drivers are consumers of TDI. They use the TDI API to communicate with TDI Transport drivers. AFD.sys, NETBT.sys and HTTP.sys are examples of TDI Clients.
- TDI Transport drivers are service providers for TDI. They typically implement transport and network layer functionality. They handle calls made by TDI clients and provide access to the network adapter through NDIS. Consequently most TDI transports are also NDIS protocol drivers. TCPIP.sys, NWLINK.sys and TCPIP6.sys are examples of TDI Transports.
- TDI Filter drivers sit between TDI clients and TDI transports and intercept the communication between them. Most network firewall drivers on Windows are implemented as TDI Filters.

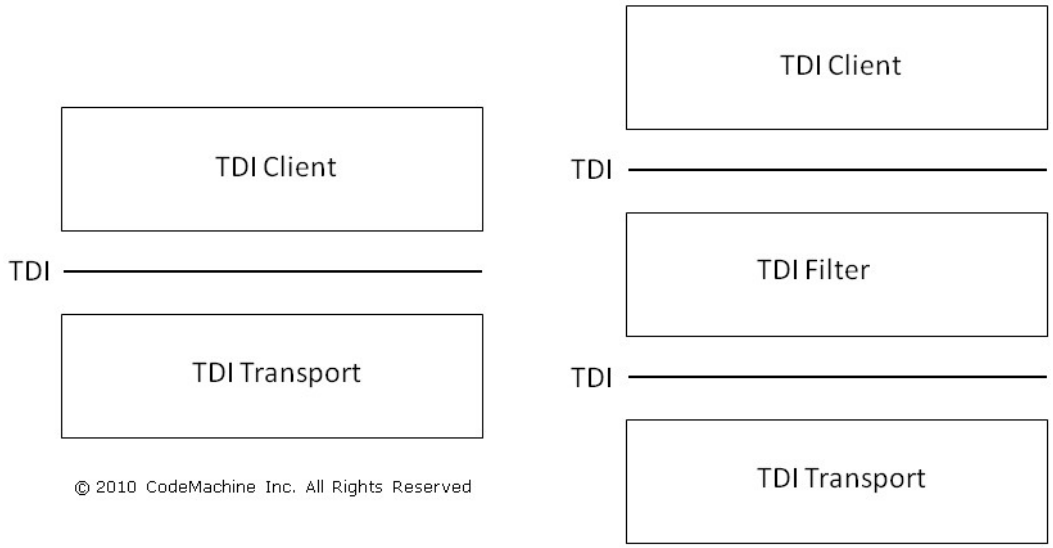

Figure 1 : TDI Driver Types

Although, TDI Clients typically sit above TDI transports, some TDI clients can be logically at the same level or below TDI transports in the network stack. Here are some examples of such TDI Client drivers :

- TCPIPv6.sys is a TDI transport driver, which also acts as a TDI client since it uses TCPIP.sys to tunnel IPv6 traffic over IPv4.
- RASPPTP.sys is an NDIS miniport driver and a TDI client since it uses TCPIP.sys to tunnel PPTP traffic over UDP.
- RASL2TP.sys is an NDIS miniport driver and a TDI client which uses TCPIP.sys to tunnel L2TP/IPSEC traffic over UDP.

The Windows Driver Kit (including the Windows 7 WDK) contains the necessary header files and libraries that can be used to build TDI drivers. Although the Windows 7 WDK does not contain any TDI sample code, TDI client driver samples can be found in earlier versions of the Windows Driver Kit.

The following TDI header files, available in the WDK, can be used to build TDI drivers :

- inc\api\tdi.h
- inc\api\tdiinfo.h
- inc\api\tdikrnl.h
- inc\api\tdistat.h

TDI Drivers link to the export library tdi.lib which calls into the export driver tdi.sys.

TDI Objects

The TDI specification is based around I/O manager data structure like IRPs, device objects and files objects, which most Windows kernel developers are already familiar with. These data structures are used extensively by TDI drivers. For example, TCPIP.sys uses device objects to represent various types of socket as shown in the table below :

Device Name	Socket Type
\Device\Ip	ICMP
\Device\RawIp	Raw
\Device\Tcp	Stream
\Device\Udp	Datagram
\Device\IPMULTICAST	IGMP

TDI Filter Drivers attach above one or more of the device objects created by TCPIP.sys in order to filter network traffic at the transport level.

TDI client drivers use file objects to represent open sockets as shown in the table below :

- Address Object which represents the local endpoint of a bound socket. These objects are used both for TCP and UDP sockets.
- Connection Object which represents the remote endpoint of a connected socket and is used for TCP sockets only.
- Control Channel Object is used for network management to query or set global configuration and management information for a service provider.

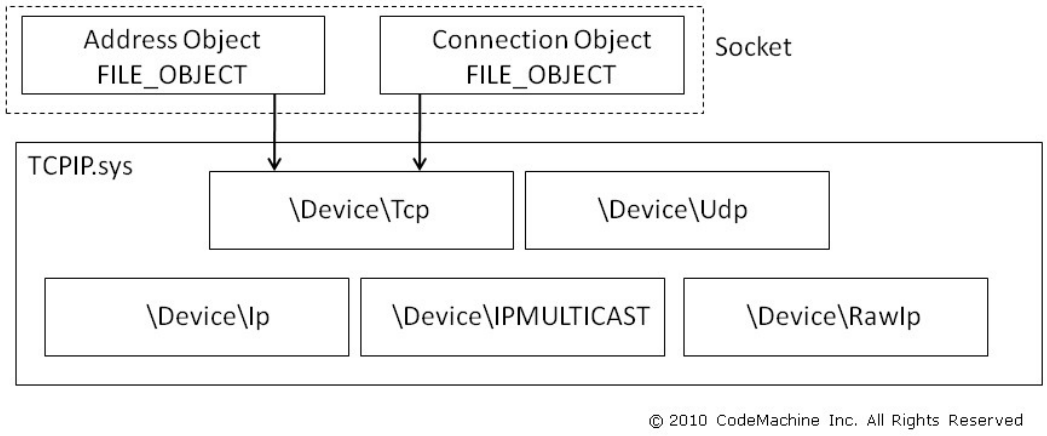

Figure 2 : TCP Socket consisting of Address and Connection File Object

TDI Clients call the I/O Manager API ZwCreateFile() to create file objects. The Extended Attributes (EA) information (i.e. EaName and EaValue) passed to the ZwCreateFile() API determines the type of file object that would be created :

Object Type	EaName	EaValue
Address	"TdiTransportAddress"	TA_IP_ADDRESS
Connection	"TdiConnectionContext"	PVOID

Control Channel	None	None

TDI Transport Addresses

Transport addresses (both local and remote) are expressed as TRANSPORT_ADDRESS structures which are used for all types of addresses e.g. IP, IPX, NetBIOS etc and are defined as follows:

```
typedef struct _TRANSPORT_ADDRESS {
    LONG  TAAddressCount;
    TA_ADDRESS  Address[1];
} TRANSPORT_ADDRESS, *PTRANSPORT_ADDRESS;
```

A specific type of address i.e. IP, IPX, NetBIOS is expressed as a TA_ADDRESS structure. This is defined as follows:

```
typedef struct _TA_ADDRESS {
    USHORT  AddressLength;
    USHORT  AddressType;
    UCHAR   Address[1];
} TA_ADDRESS, *PTA_ADDRESS;
```

The following table shows the information used to populate the individual fields of the TA_ADDRESS structure :

Field	Content
TA_ADDRESS.AddressLength	Set to TDI_ADDRESS_LENGTH_IP
TA_ADDRESS.AddressType	Set to TDI_ADDRESS_TYPE_IP
TA_ADDRESS.Address	Contains the TDI_ADDRESS_IP structure

And TDI_ADDRESS_IP is defined similar to the sockaddr_in structure :

```
typedef struct _TDI_ADDRESS_IP {
    USHORT  sin_port;
    ULONG  in_addr;
    UCHAR  sin_zero[8];
} TDI_ADDRESS_IP, *PTDI_ADDRESS_IP;
```

All TDI functions that deal with IP Address and Port information use this format. Addresses are always stored in network byte order.

TDI Operations

TDI clients communicate with TDI transports using requests (IRPs) and event (callbacks). Requests are initiated by TDI clients but processed and completed by TDI transports. Events are callbacks that are registered by TDI clients with TDI transports which the transports invoke when they need to notify

clients about something. The following figure shows all the possible interactions between TDI clients and TDI transports.

Figure 3 : TDI IRPs and Events

A TDI client can solicit event notifications from a TDI transport using the following two mechanisms:

- Sending a TDI IRP (request) to the TDI transport driver. The TDI transport defers this IRP by returning STATUS_PENDING and completes it at a later point in time when the specified event occurs, causing the TDI client's I/O completion routine to be invoked.
- Registering a TDI callback (event) with the TDI transport driver. The TDI transport invokes this callback function when the specified event occurs.

A TDI client can use both requests and events simultaneously. For example a TDI client can send a TDI_RECEIVE IRP and register a TDI_EVENT_RECEIVE callback both of which are used to retrieve data from the transport once it had arrived from the network.

TDI IRPs

TDI Interface uses IRPs to represent requests made from clients to transports. TDI specifies how IRPs must be formatted for clients to be able to communicate with transports. Kernel Mode TDI client drivers use internal device I/O control IRPs i.e. IRPs with the major code of IRP_MJ_INTERNAL_DEVICE_CONTROL to send TDI_xxx requests to TDI transport drivers. The I/O stack locations for such IRPs are formatted as TDI_REQUEST_KERNEL_xxx structures that contain the request specific parameters. The TDI library wrapper provides the following macros to help drivers with allocating and building TDI IRPs :

- TdiBuildInternalDeviceIoControlIrp() which internally calls the kernel API

IoBuildDeviceIoControlIrp(). Such IRPs should not be freed using IoFreeIrp(). When IoCompleteRequest() is called the IRP is automatically freed after the necessary completion routines have been invoked.

- TdiBuildxxx(), where xxx represents any of the internal device I/O control requests, populates the TDI_REQUEST_KERNEL_xxxx data structures, that are passed to the TDI transport driver, as a part of the TDI request.

The following table lists the TdiBuildxxx() macros that can be used by TDI client drivers to format TDI requests.

Request	Build Function	AFD/TCPIP Usage
TDI_ASSOCIATE_ADDRESS	TdiBuildAssociateAddress()	Associate an address and connection object for a TCP socket.
TDI_DISASSOCIATE_ADDRESS	TdiBuildDisassociateAddress()	Break the association between an address and a connection object that were previously associated with each other.
TDI_CONNECT	TdiBuildConnect()	Initiate a TCP 3-way setup handshake (SYN, SYN+ACK, ACK) using a particular connection object.
TDI_LISTEN	TdiBuildListen()	Solicit notifications for inbound SYNs on a particular address object.
TDI_ACCEPT	TdiBuildAccept()	Request the TCP stack to respond to an inbound SYN with a SYN+ACK using a particular connection object.
TDI_DISCONNECT	TdiBuildDisconnect()	Initialize a TCP 3-way teardown (FIN, FIN+ACK, ACK) handshake with a remote system.
TDI_SEND	TdiBuildSend()	Transmit stream data using the specified connection object.
TDI_RECEIVE	TdiBuildReceive()	Request TCP to return data received on a specific connection object.

TDI_SEND_DATAGRAM	TdiBuildSendDatagram()	Transmit datagram packet(s) using the specified address object.
TDI_RECEIVE_DATAGRAM	TdiBuildReceiveDatagram()	Request UDP to return data received on a specific address object.
TDI_SET_EVENT_HANDLER	TdiBuildSetEventHandler()	Register/Unregister event callbacks for a given address object with TCPIP.sys.
TDI_QUERY_INFORMATION	TdiBuildQueryInformation()	Read Management Information Base (MIB) information from the TCP/UDP/IP stack.
TDI_SET_INFORMATION	TdiBuildSetInformation()	Not Supported by TCPIP.sys.
TDI_ACTION	TdiBuildAction()	Not Supported by TCPIP.sys.

TDI Filters can intercept any of the above IRPs to perform value added filtering. Although most TDI clients (including AFD.sys) use IRPs to send requests to TDI transports there are some exceptions. For performance reasons netbt.sys, a TDI client driver, retrieves a pointer to an internal function in TCPIP.sys using the IOCTL code IOCTL_TDI_QUERY_DIRECT_SEND_HANDLER. It then calls this function (i.e. TCPSendData()), directly, instead of building an IRP and sending it to TCPIP.sys, as a TDI client would normally do. Filter drivers need to be aware of this behavior when attempting to intercept SMB (Windows Print and File Sharing) traffic.

The TDI wrapper library also provides an API (i.e. TdiMapUserRequest()) to translate device I/O control TDI requests arriving from user mode into internal device I/O control requests.

TDI Events

Registering events with TDI transport drivers alleviates the need for clients to have requests (IRPs) pending with TDI transports at all times, to receive notifications about events. Event callbacks also optimize memory usage for incoming data, since the transport driver can use buffers indicated by underlying network drivers to pass the data directly to TDI clients. TDI Clients can solicit specific TDI event indications by registering event callbacks and callback contexts for a specific address object, using the internal device I/O control request TDI_SET_EVENT_HANDLER. The following table lists all the events that are supported by TDI and indicates which ones are used by AFD.sys and TCPIP.sys :

Event	**AFD/TCP Usage**
TDI_EVENT_CONNECT	Used to indicate inbound TCP SYN segments to AFD.sys.

TDI_EVENT_DISCONNECT	Used to indicate inbound TCP FIN segments to AFD.sys.
TDI_EVENT_ERROR	Not used by TCPIP.sys.
TDI_EVENT_RECEIVE	Used to indicate inbound TCP data segments to AFD.sys.
TDI_EVENT_RECEIVE_DATAGRAM	Used to indicate inbound UDP datagrams to AFD.sys.
TDI_EVENT_RECEIVE_EXPEDITED	Used to indicate inbound TCP segments that have the TCP URGENT flag set to AFD.sys.
TDI_EVENT_SEND_POSSIBLE	Not used by TCPIP.sys.
TDI_EVENT_CHAINED_RECEIVE	Used to indicate data that is described directly by a NDIS_BUFFER (i.e. MDL chain) as opposed to a TCPIP buffered copy of the data.
TDI_EVENT_CHAINED_RECEIVE _DATAGRAM	Not used by TCPIP.sys.
TDI_EVENT_CHAINED_RECEIVE_EXPEDITED	Not used by TCPIP.sys.
TDI_EVENT_ERROR_EX	Used by TCPIP to indicate destination host unreachable error events to AFD.sys.

In order for TDI Filter drivers to intercept TDI event callbacks, they have to first intercept the TDI_SET_EVENT_HANDLER request, save the original callback function pointers that are being registered by the TDI client and then replace the callbacks with its own function pointers. This causes the underlying transport to invoke the TDI filters callbacks directly instead of the TDI client callbacks. When these callbacks get called, the TDI filter would have to invoke the original callback to propagate the event notification to the TDI client.

Most event handlers are called at DISPATCH_LEVEL, so the TDI client cannot block in these handlers. Some event handlers can return STATUS_MORE_PROCESSING_REQUIRED along with a TDI request IRP thus indicating to the underlying transport that the event processing can be continued using the TDI request.

TDI Query/Set Information

The TCPIP stack allows both user and kernel mode clients to query statistics and set management information related to various entities in the stack i.e. TCP, UDP, IP and ICMP.

User mode clients use the device I/O control code IOCTL_TCP_QUERY_INFORMATION_EX to query information from the TCPIP stack and IOCTL_TCP_SET_INFORMATINON_EX to set information into

the TCPIP stack. Kernel mode clients can query information from the TCPIP stack using the internal I/O control request code TDI_QUERY_INFORMATION.

These IOCTLs are partially documented in the Platform SDK, however Microsoft strongly discourages their usage as they may be modified or removed in the future.

The TCP_REQUEST_QUERY_INFORMATION_EX and the TCP_REQUEST_SET_INFORMATION_EX structures are used by user mode clients to specify the parameters for the query and set requests. Both these structures contain the TDIObjectID structure that identifies the type and instance of information being queried or set as shown below:

```
typedef struct {
    TDIEntityID toi_entity;
    unsigned long toi_class;
    unsigned long toi_type;
    unsigned long toi_id;
} TDIObjectID;
```

```
typedef struct {
    unsigned long tei_entity;
    unsigned long tei_instance;
} TDIEntityID;
```

For example in order to disable the TCP stack's use of Nagle algorithm, user mode clients setup the object ID structure as follows:

```
TDIObjectID.toi_entity.tei_entity       =       CO_TL_ENTITY;
TDIObjectID.toi_entity.tei_instance     =       TL_INSTANCE;
TDIObjectID.toi_class                   =       INFO_CLASS_PROTOCOL;
TDIObjectID.toi_type                    =       INFO_TYPE_CONNECTION;
TDIObjectID.toi_id                      =       TCP_SOCKET_DELAY;
```

The actual data values that are queried and set using the IOCTLs are packaged in the form of Simple Network Management Protocol (SNMP) MIB structures. This allows access to functionality like Adapter Management, MAC Address Resolution, Interface Management, Routine Table Manipulation and Stack Management.

The IP Helper API DLL (IPHLPAPI.DLL), which is used by networking utilities like netstat, tracert, ipconfig, route and arp also use these IOCTLs to query information and set parameters in the network stack. Most IPHLPAPI APIs call the Window Native API NtDeviceIoControlFile() on a file handle obtained to the device "\Device\Tcp" with the above mentioned I/O control codes.

Client and Server Side Sockets

Sockets have 2 endpoints, a local endpoint and a remote one. An address object in TDI represents the local endpoint of a bound socket. A connection object represents the remote endpoint of a connected socket.

For client side sockets there is one connection object associated with one address object for each

outbound connection that is established, as shown on the left hand side of the figure below. For server side sockets there are as many connection objects associated with a single address object, as there are connections established by clients to that server port, as shown in right hand side of the following figure :

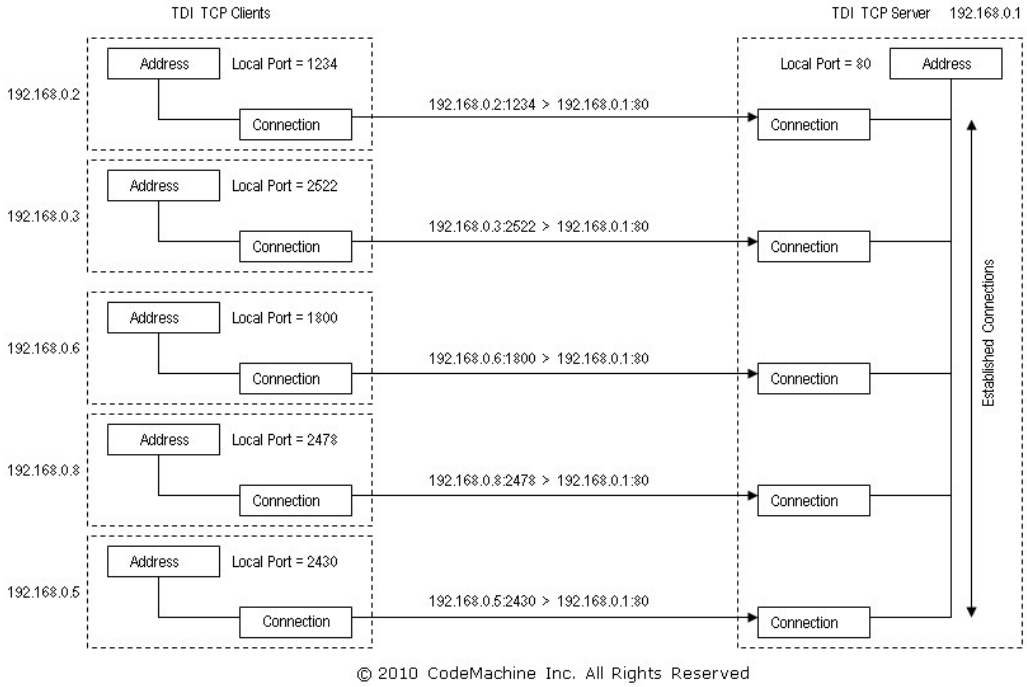

Figure 4 : TCP Client and Server side Address and Connection Objects

Each incoming client connection request, whose destination port matches the port number on the address object, is assigned a connection object out of the pool of connection objects maintained by the TDI transport. Each such connection object represents a socket, that has been accepted by the server for data transfers.

Client side TDI interactions

TCP Clients perform the following steps to setup an outbound TCP connection :

- Create an transport address object (ZwCreateFile())
- Create a connection endpoint object (ZwCreateFile())
- Associate the transport address object with the connection endpoint object (TDI_ASSOCIATE_ADDRESS)
- Request a connection setup to the remote system (TDI_CONNECT)

TCP Clients perform the following steps to teardown a TCP connection :

- Request a graceful or abortive connection teardown to the remote system (TDI_DISCONNECT)
- Break the association of with the connection endpoint object with the transport address object (TDI_DISASSOCIATE_ADDRESS)
- Close the transport address object (ZwClose())
- Close the connection endpoint object (ZwClose())

The following figure shows the various interactions that a TDI client has with the a TDI transport for a client side (connection) socket :

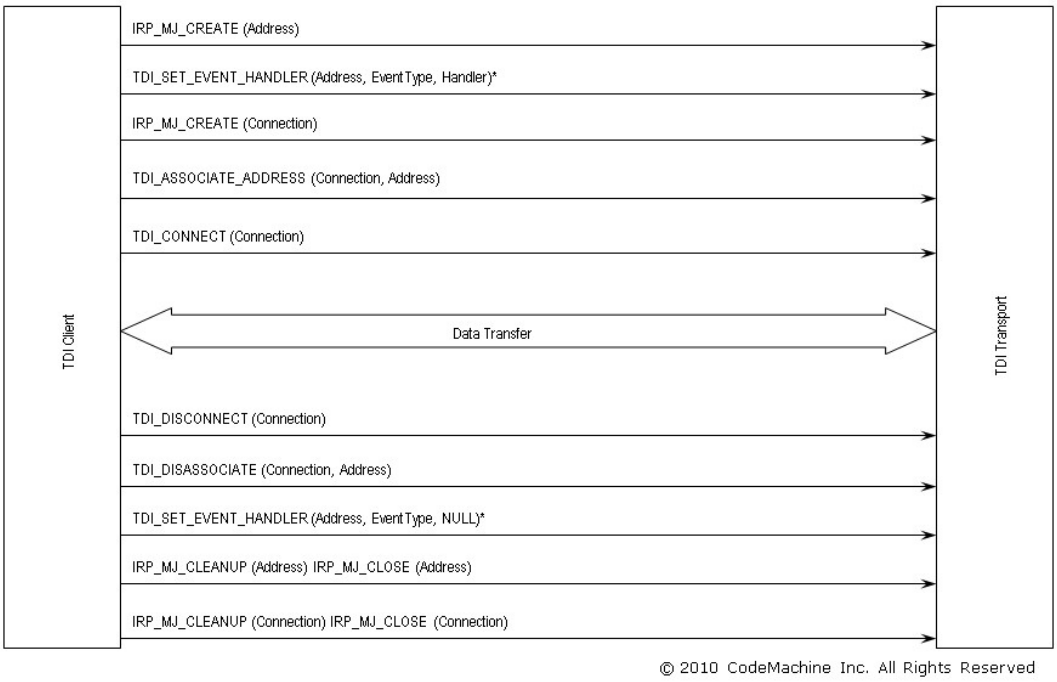

Figure 5 : Client side socket setup and teardown

Server side TDI interactions

TCP Servers perform the following steps to setup a server side socket and accept an inbound TCP connection :

- Create an transport address object (ZwCreateFile())
- Register a connection event handler (TDI_SET_EVENT_HANDLER + TDI_EVENT_CONNECT)
- Create a set of connection endpoint objects (ZwCreateFile())
- Associate the transport address object with all the connection endpoint objects (TDI_ASSOCIATE_ADDRESS)
- When the connection event handler is invoked due to an inbound connection request, examine the remote address information, select an available connection endpoint object and request a connection acceptance (TDI_ACCEPT)

The following figure shows the various interactions that TDI client has with the TDI transport for a server side (listening) socket:

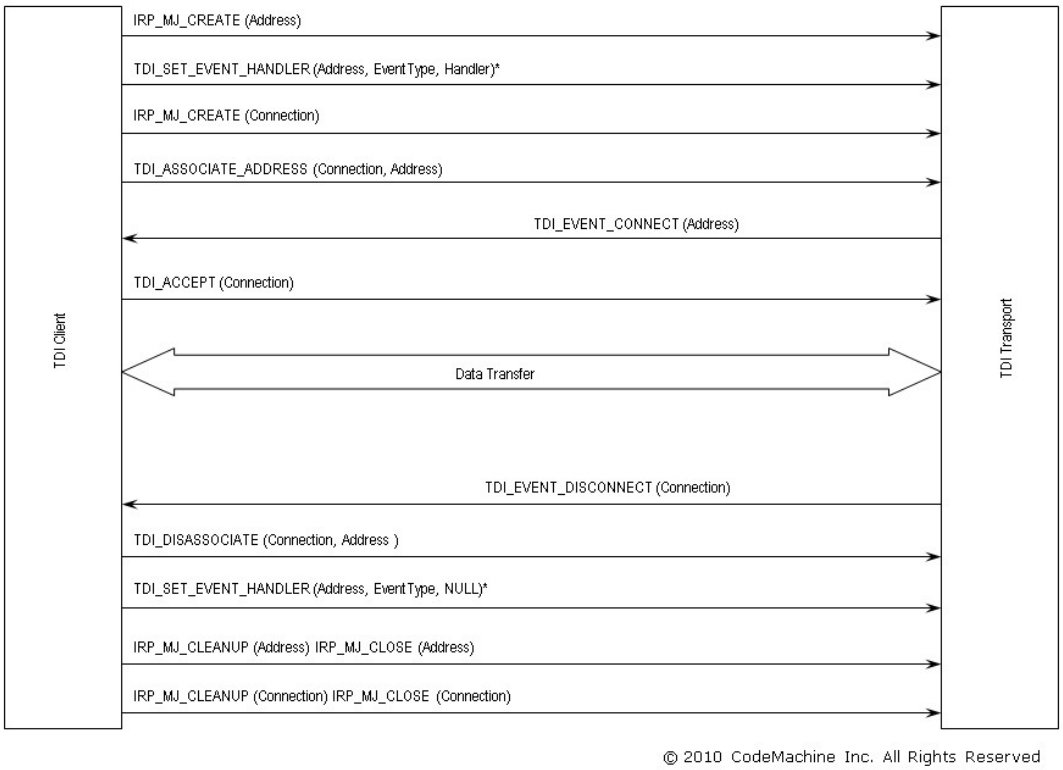

Figure 6 : Server side socket setup and teardown

TDI Data Transfers

TCPIP.sys supports the data transfer request IRPs TDI_SEND, TDI_RECIVE and TDI_RECEIVE_DATAGRAM and the data transfer callbacks TDI_EVENT_RECEIVE, TDI_EVENT_RECEIVE_DATAGRAM, TDI_EVENT_RECEIVE_EXPEDITED and TDI_EVENT_CHANINED_RECEIVE.

Buffers used by these TDI data transfers are described either by Memory Descriptor List (MDLs) or by raw pointers to the data itself. MDLs are allocated by the caller and are attached to the TDI request IRP at IRP->MdlAddress. Depending on the layout of the data buffer used in a rquest, a single request can contain multiple MDLs. Multiple MDLs can be chained together in the form of a single linked list through the MDL->Next field. For instance, a user mode WinSock client calling the API TransmitFile(), causes AFD.sys to send a TDI_SEND request with chained MDLs to TCPIP.sys, wherein each MDL points to a virtually contiguous buffer.

When receiving data, TDI client drivers that use event callbacks, can flow off TDI Transport drivers by returning STATUS_DATA_NOT_ACCEPTED from their receive event callback. This causes the TDI transport to retain the data and stop any further receive data indications for that connection. To resume the receive data indication and retrieve any queued data, the TDI client can subsequently send a TDI_RECEIVE IRP to the underlying transport driver.

Similarly, TDI client drivers that use requests, can flow off the TDI transport by temporarily suspending the queuing of further TDI_RECIVE requests, to the underlying TDI transport.

Flowing off stream oriented transports (like TCP), causes the TCP stack to advertise a reduced window

size to its remote peer, thus flowing off data transmission from the remote system.

The TDI wrapper library (tdi.sys) provides APIs like TdiCopyBufferToMdl(), TdiCopyMdlToBuffer(), TdiCopyLookaheadData() and TdiCopyMdlChainToMdlChain(), to help client and transport drivers with data copying operations.

AFD/TCPIP Interface

The following figure, shows the names of the internal functions in AFD.sys that serve as TDI event handlers which are invoked by TCPIP.sys. It also shows the internal components of TCPIP.sys that process TDI requests and the functions that serve as entry points into these components. Knowing these function names helps in setting breakpoints during live debugging. Note, that these functions names apply to Windows XP and Windows Server 2003 only.

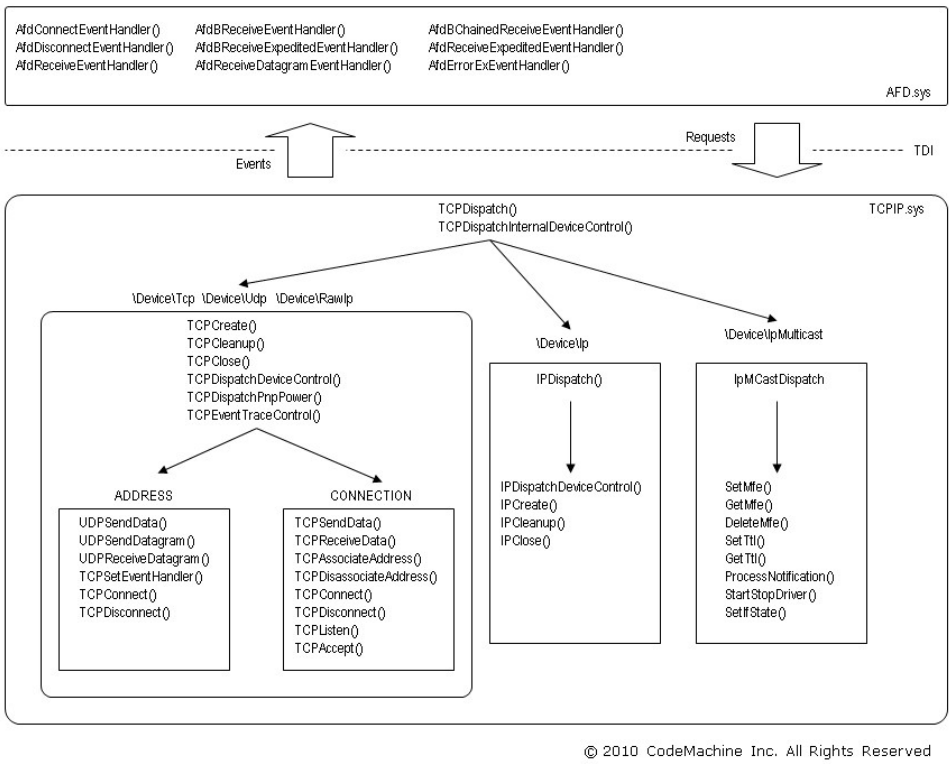

Figure 7 : TCPIP.sys request handlers and AFD.sys event handlers

TCPIP.sys installs 2 dispatch entry points in the driver object, one for internal device I/O control (TCPDispatchInternalDeviceControl()) and another for all other IRPs (TCPDispatch()). The internal device I/O control entry point is the most frequently exercised dispatch routine inside TCPIP.sys.

Depending on the particular device object an IRP is targeted at, TCPIP.sys will route the request to appropriate internal dispatch routines. Additionally the FILE_OBJECT->FsContext2 field of TCPIP.sys file objects contain a magic number that enables TCPIP.sys to route requests to a particular set of routines e.g. Address or Connection processing routines, within TCPIP.sys.

TDX

Starting with Windows Vista, Microsoft has made several attempts to remove the support for TDI drivers from the operating system. This would have resulted in all legacy TDI clients and TDI filters becoming non-functional on subsequent versions of Windows. Although industry pressure on Microsoft has prevented this from happening so far, it is bound to happen eventually. Since TDI is on the path of deprecation, the windows networking stack provides new technologies that replace TDI, which developers are encouraged to adopt. Drivers on Vista and later versions of Windows that need to implement TDI client functionality should use the Windows Socket Kernel (WSK) interface and drivers that need to implement TDI filtering functionality should use Windows Filtering Platform (WFP) interface.

Due to the re-architecture of the networking stack in Vista, TDI is no longer the interface that AFD.sys uses to communicate with TCPIP.sys. Instead, AFD.sys uses a new undocumented interface called Transport Layer Network Provider Interface (TLNPI) to communicate with TCPIP.sys. However, in order to support legacy TDI clients and TDI filters, Microsoft provides a new driver called TDX.sys, which internally use TLNPI to communicate with TCPIP.sys. It also creates all the device objects that TCPIP used to create, in order to maintain backward compatibility with legacy TDI drivers. The figure below shows the relationship between the components mentioned above on Vista and later versions of Windows. When Windows detects the presence of TDI filter in the system, all traffic between AFD.sys and TCPIP.sys is automatically routed through the TDX driver. The TDX driver makes use of the TDI API in TDI.sys and uses the Network Module Registrar (NMR) API in NETIO.sys to implement its functionality. TDX is supported on Vista, Server 2008 and Windows 7. TDX handles TDI requests from legacy TDI drivers and maps them to TLNPI calls.

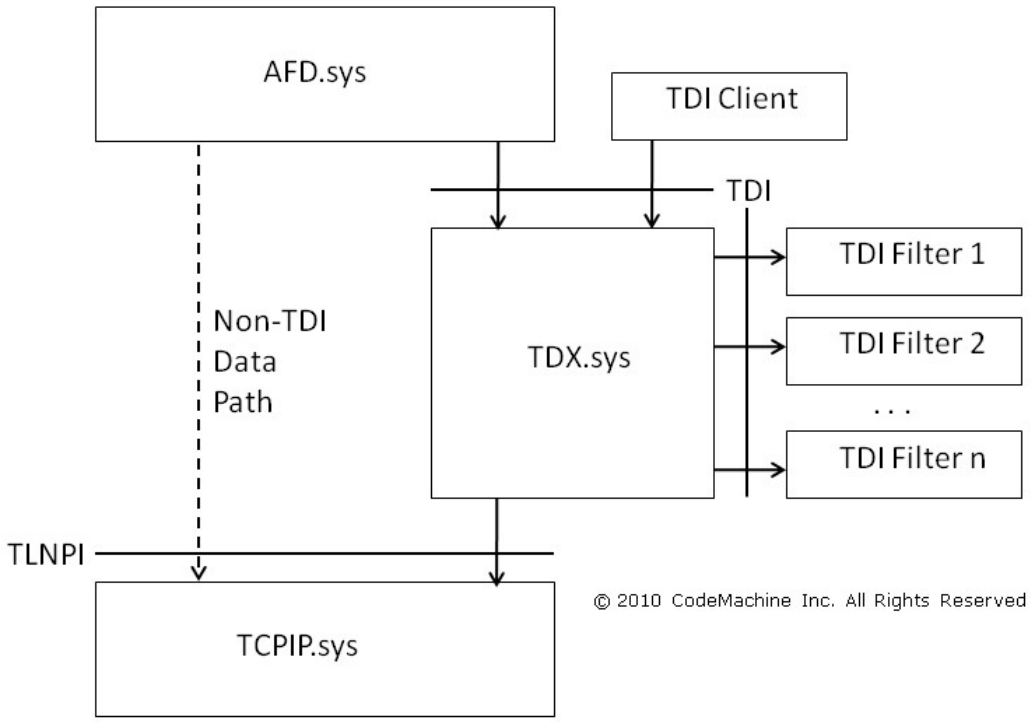

Figure 8 : TDX, TDI Clients and TDI Filters on Windows Vista and later versions of Windows.

Since the inception of Windows NT, TDI has played a critical role in the Windows networking stack. TDI

had standardized the interface between clients and transports in the networking stack, thus enabling development of kernel mode socket clients and filters that have added significant value to the Windows networking stack. TDI has now outlived its usefulness and will be removed from the networking stack sooner or later paving the way for newer and more robust interfaces like WFP and WSK.
