TCPDirect User Guide

SF-116303-CD , Issue 4

2017/06/28 12:04:32

Solarflare Communications Inc

TCPDirect User Guide

SF-116303-CD

Last Revised: 62017

Issue 4

Contents

Chapter 1

TCPDirect

Solarflare's TCPDirect is highly accelerated network middleware. It uses similar techniques to Onload, but delivers lower latency. In order to achieve this, TCPDirect supports a reduced feature set and uses a proprietary API.

1.1 Introduction

The TCPDirect API provides an interface to an implementation of TCP and UDP over IP. This is dynamically linked into the address space of user-mode applications, and granted direct (but safe) access to the network-adapter hardware. The result is that data can be transmitted to and received from the network directly by the application, without involvement of the operating system. This technique is known as 'kernel bypass'.

Kernel bypass avoids disruptive events such as system calls, context switches and interrupts and so increases the efficiency with which a processor can execute application code. This also directly reduces the host processing overhead, typically by a factor of two, leaving more CPU time available for application processing. This effect is most pronounced for applications which are network intensive.

The key features of TCPDirect are:

- **User-space**: TCPDirect can be used by unprivileged user-space applications.
- **Kernel bypass**: Data path operations do not require system calls.
- **Low CPU overhead**: Data path operations consume very few CPU cycles.
- **Low latency**: Suitable for low latency applications.
- **High packet rates**: Supports millions of packets per second per core.
- **Zero-copy**: Particularly efficient for filtering and forwarding applications.
- **Flexibility**: Supports many use cases.

Chapter 2

Overview

This part of the documentation gives an overview of TCPDirect and how it is often used.

2.1 Platforms

TCPDirect can be run on 8000-series Solarflare adapters with a suitable license (e.g. the 'Plus' license).

TCPDirect can also be run on 7000-series adapters which require both the Onload license and a TCPDirect license.

Refer to the *Solarflare Server Adapter User Guide* 'Product Specifications' for adapter details; for licensing queries, please contact your sales representative.

TCPDirect is supported on:

- Red Hat Enterprise Linux 6.6 - 6.8
- Red Hat Enterprise Linux 7.0 - 7.2
- SuSE Linux Enterprise Server 11 sp2, sp3, sp4
- SuSE Linux Enterprise Server 12 base and sp1
- Canonical Ubuntu Server LTS 14.04, 16.04
- Canonical Ubuntu Server 16.10
- Debian 7 "Wheezy"
- Debian 8 "Jessie"
- Linux kernels 2.6.32 - 4.8.

2.2 Components

TCPDirect is supplied as:

- header files containing the proprietary public API
- a binary library for linking into your application.

To use TCPDirect, you must have access to the source code for your application, and the toolchain required to build it. You must then replace the existing calls for network access with appropriate calls from the TCPDirect API. Typically this involves replacing calls to the BSD sockets API. Finally you must recompile your application, linking in the TCPDirect library.

For more details, see Using TCPDirect.

If you do not have access to source code for your application, you can instead accelerate it with Onload.

2.3 Capabilities and Restrictions

TCPDirect supports a carefully selected feature set that allows it to run many real-world applications, without losing performance to resource-intensive features that are seldom used.

Before porting an application to TCPDirect, you should ensure that it supports the features that you require. The subsections below list the support for different features.

If your application requires features that are unsupported by TCPDirect, consider instead using Onload or ef_vi:

- Onload has higher latency than TCPDirect, but a full feature set.

  Onload supports all of the standard BSD sockets API, meaning that no modifications are required to POSIX-compliant socket-based applications being accelerated. Like TCPDirect, Onload uses kernel bypass for applications over TCP/IP and UDP/IP protocols.

- Ef_vi has even lower latency than TCPDirect, but operates at a lower level.

  Ef_vi is a low level OSI level 2 interface which sends and receives raw Ethernet frames, and exposes many of the advanced capabilities of Solarflare network adapters. But because the ef_vi API operates at this low level, any application using it must implement the higher layer protocols itself, and also deal with any exceptions or other unusual conditions.

2.3.1 Protocols

The table below shows the protocols that are supported by TCPDirect and (for comparison) by Onload:

Protocol	TCPDirect	Onload
IPv4	Yes	Yes
IPv6	No	No

UDP	Yes	Yes
TCP	Yes	Yes
TCP header options (e.g. timestamps)	No	Yes
VLANs	Yes	Yes
Multicast RX	Yes	Yes
Multicast TX	Yes	Yes
Multicast loopback	No	Yes

2.3.2 OS

The table below shows the OS features that are supported by TCPDirect and (for comparison) by Onload:

OS	TCPDirect	Onload
Preload	No	Yes
Static link	Yes	Yes
Dynamic link	Yes	Yes
Direct API	Yes	Yes
Bonding	No	Yes
Teaming	No	Yes
Send/receive via non-SFC interface	No	Yes
Multiple threads	Yes	Yes
Multiple processes	Yes	Yes
Sharing stacks between threads and processes	No	Yes
Multiple stacks	Yes	Yes
fork()	Yes, with limitations (no shared stacks or zockets)	Yes
dup()	N/A (no file descriptors)	Yes
User-level only	Yes	No
Interrupts	No	Yes
Huge pages	Yes	Yes

As mentioned in the table above, TCPDirect stacks and zockets cannot be shared between processes. The one partial exception is that a stack may be used in a `fork()` child if it is not used in the parent after calling `fork()`.

2.4 How TCPDirect Increases Performance

TCPDirect can significantly reduce the costs associated with networking by reducing CPU overheads and improving performance for latency, bandwidth and application scalability.

2.4.1 Overhead

Transitioning into and out of the kernel from a user-space application is a relatively expensive operation: the equivalent of hundreds or thousands of instructions. With conventional networking such a transition is required every time the application sends and receives data. With TCPDirect, the TCP/IP processing can be done entirely within the user-process, eliminating expensive application/kernel transitions, i.e. system calls. In addition, the TCPDirect TCP/IP stack is highly tuned, offering further overhead savings.

The overhead savings of TCPDirect mean more of the CPU's computing power is available to the application to do useful work.

2.4.2 Latency

Conventionally, when a server application is ready to process a transaction it calls into the OS kernel to perform a 'receive' operation, where the kernel puts the calling thread 'to sleep' until a request arrives from the network. When such a request arrives, the network hardware 'interrupts' the kernel, which receives the request and 'wakes' the application.

All of this overhead takes CPU cycles as well as increasing cache and translation lookaside-buffer (TLB) footprint. With TCPDirect, the application can remain at user level waiting for requests to arrive at the network adapter and process them directly. The elimination of a kernel-to-user transition, an interrupt, and a subsequent user-to-kernel transition can significantly reduce latency. In short, reduced overheads mean reduced latency.

2.4.3 Bandwidth

Because TCPDirect imposes less overhead, it can process more bytes of network traffic every second. Along with specially tuned buffering and algorithms designed for high speed networks, TCPDirect allows applications to achieve significantly improved bandwidth.

2.4.4 Scalability

Modern multi-core systems are capable of running many applications simultaneously. However, the advantages can be quickly lost when the multiple cores contend on a single resource, such as locks in a kernel network stack or device driver. These problems are compounded on modern systems with multiple caches across many CPU cores and Non-Uniform Memory Architectures.

TCPDirect results in the network adapter being partitioned and each partition being accessed by an independent copy of the TCP/IP stack. The result is that with TCPDirect, doubling the cores really can result in doubled throughput.

2.5 Requirements

2.5.1 Adapter

The following list identifies the minimum driver and firwmare requirements for adapters running TCPDirect applications:

- Net driver minimum: 4.10.1011 - this is available in the openonload-201606-u1 and Enterprise Onload 5.0 distributions.
- Firmware minimum: 6.2.3.1000
- Firmware variant: The adapter must be configured to use the ultra-low-latency firmware variant. The firmware variant can be identified/set using the Solarflare sfboot utility from the Solarflare Linux Utilities package (SF-107601-LS).

2.5.2 License

SFN8000 series adapters - require an Onload license and TCPDirect license. The 'Plus' license will include both required licenses.

SFN7000 series adapters - require an Onload license and TCPDirect license.

Installed licenses can be identified using the Solarflare sfkey utility from the Solarflare Linux Utilities package (SF-107601-LS).

```
# sfkey
enp4s0f0,enp4s0f1: 712200205071234567890123 (Flareon)
  Product name          Solarflare SFN7122F SFP+ Server Adapter
  Serial number         712200205071234567890123
  Installed keys        Onload, TCP Direct

enp5s0f0,enp5s0f1: 000F5341C700 (8xxx)
  Product name          Solarflare Flareon Ultra 8000 Series 10G Adapter
  Serial number         852200203001234567890123
  Installed keys        Plus
```

2.5.3 Onload

Openonload from version 201606-u1 or EnterpriseOnload 5.0 must be installed on the server.

2.5.4 Huge Pages

TCPDirect requires the allocation of huge pages. Huge pages are needed for each stack created - approximate mininum 10 huge pages per stack, the number of zockets created and number of packet buffers required. Some experimentation is needed to identify suitable page allocation needs for an application, but a general recommendation would be to allocate at least 40 huge pages per stack and then to use zf_stackdump to identify packet buffer usage.

Further information including allocation commands for huge pages is available in the Onload User Guide (SF-104474-CD).

2.5.5 PIO

TCPDirect uses PIO packet buffers and these are available by default from the adapter driver. Users should be aware that PIO buffers are a limited resource used by the driver for non-accelerated sockets, by Onload stacks which require 1 PIO buffer per VI created and by the TCPDirect application which require 1 PIO buffer per stack.

To ensure there are sufficient PIO buffers available, it may be necessary to restrict or prevent the driver and Onload non-critical sockets from using PIO.

Further information about PIO including configuration commands can be found in the Onload User Guide (SF-104474-CD).

Chapter 3

Concepts

This part of the documentation describes the concepts involved in TCPDirect.

3.1 Stacks

TCPDirect can have multiple network stacks. Each stack accesses a separate partition of the network adaptor, which improves security. Access to a given stack is not thread-safe, so typically each thread uses a separate TCPDirect stack. This model avoids problems of lock-contention and cache-line bouncing and allows for scalability in a natural way.

3.2 Zockets

TCPDirect endpoints are represented by *zockets*. Zockets are similar to BSD or POSIX sockets, but are categorized by protocol and state into different zocket types.

Each type of zocket is represented by its own distinct data structure, and is handled by its own API calls. Because the type of zocket is known, code can be more efficient.

Zockets cannot be converted from one type to another.

3.2.1 TCP zockets

For TCP, a new zocket can be listening or non-listening:

- a listening zocket is represented by a *TCP listening zocket* data structure
- a non-listening zocket is represented by a *TCP zocket handle*.

If the zocket is later connected, a *TCP zocket* data structure is created and returned. This occurs if:

- a connection to a TCP listening zocket is accepted
- a TCP zocket handle is connected.

3.2.2 UDP zockets

For UDP, a zocket can be used to receive or transmit:

- a receive zocket is represented by a *UDP receive zocket* data structure
- a transmit zocket is represented by a *UDP transmit zocket* data structure

3.2.3 Waitables

Waitables are handles that represent zockets and are used with the multiplexer interface.

Each type of zocket has an API call to return a waitable representing the zocket.

3.3 Multiplexers

A *multiplexer* (or *muxer*) allows multiple zockets to be polled for activity through a single call. The interface and behaviour are similar to the standard Linux epoll mechanism.

Each multiplexer is associated with a stack and can only be used to poll zockets from that stack. Zockets (represented by waitables) can be added to a multiplexer together with the set of events that the application is interested in. A zocket can only be a member of one multiplexer at a time.

When a multiplexer is polled the corresponding stack is polled for network events, and a list of zockets that are *ready* (readable, writable etc.) is returned.

3.4 TX alternatives

TX alternatives provide multiple alternative queues for transmission, that can be used to minimize latency. Different possible responses can be pushed through the TX path on the NIC, and held in different queues ready to transmit. When it is decided which response to transmit, the appropriate alternative queue is selected, and the queued packets are sent. Becasue the packets are already prepared, and are held close to the wire, latency is greatly reduced.

Due to differences in hardware architecture, the TX alternatives feature is not available on SFN7000 series adapters.

Chapter 4

Example Applications

Solarflare TCPDirect comes with a range of example applications - including source code and make files. This is a quick guide to using them, both for testing TCPDirect's effectiveness in an environment, and as starting points for developing applications.

Application	Description
zfudppingpong	Measure round-trip latency with UDP.
zftcppingpong	Measure round-trip latency with TCP.
zfaltpingpong	Measure round-trip latency with TCP TX Alternatives.
zfsink	Receive stream of UDP datagrams and demonstrate muxer.
zftcpmtpong	Use of TCPDirect in multi-threaded applications.

4.1 zfudppingpong

The zfudppingpong application passes messages back and forth between two hosts using UDP, and uses this to measure the average round-trip latency.

4.1.1 Usage

Server:

```
export ZF_ATTR=interface=ethX
zfudppingpong pong serverhost:serverport clienthost:clientport
```

Client:

```
export ZF_ATTR=interface=ethX
zfudppingpong ping clienthost:clientport serverhost:serverport
```

where:

- *ethX* is the name of the network interface to use,
- *serverhost* and *clienthost* identify the server and client machines (e.g. `hostname` or `192.168.0.10`), and
- *serverport* and *clientport* are port numbers of your choosing on the server and client machines

There are various additional options. See the help text for details.

4.2 zftcppingpong

The zftcppingpong application passes messages back and forth between two hosts using TCP, and uses this to measure the average round-trip latency. It illustrates actively and passively opened TCP connections, and has an option to use a *muxer*.

4.2.1 Usage

Server:

```
export ZF_ATTR=interface=ethX
zftcppingpong pong serverhost:serverport
```

Client:

```
export ZF_ATTR=interface=ethX
zftcppingpong ping serverhost:serverport
```

4.3 zfaltpingpong

The zfaltpingpong application illustrates use of the TX alternatives feature, which supports lower latency sends with TCP.

Usage is as for zftcppingpong.

4.4 zfsink

The zfsink application demonstrates how to receive UDP datagrams, how to use the muxer, and the "waitable fd" mechanism for integration with other I/O and blocking.

By default it traces the calls it makes, and this can be suppressed with the -q option.

4.4.1 Usage

```
export ZF_ATTR=interface=ethX
zfsink localaddr:port
```

localadddr should be an IP address on interface *ethX*, or a multicast address. There are various additional options – run "zfsink -h" for details.

4.5 zftcpmtpong

The zftcpmtpong application demonstrates how to use TCPDirect in an application that does sends and receives on TCP sockets in separate threads.

By default it traces the calls it makes, and this can be suppressed with the -q option.

4.5.1 Usage

```
export ZF_ATTR=interface=ethX
zftcpmtpong localaddr:port
```

localadddr should be an IP address on interface *ethX*. This application accepts incoming TCP connections and waits for messages to arrive. It sends on each connection an equal number of bytes as are received (although not with the same contents).

4.6 Building the Example Applications

The TCPDirect example applications are built along with the Onload installation and should be present in the `openonload/build/gnu_x86_64/tests/zf_apps` subdirectory.

Source code for the example applications is in the `src/tests/zf_apps` subdirectory.

To rebuild the example applications use the following procedure:

```
cd openonload/scripts/
export PATH="$PWD:$PATH"
cd ../build/gnu_x86_64/tests/zf_apps/
make clean
make
```


Chapter 5

Using TCPDirect

This part of the documentation gives information on using TCPDirect to write and build applications.

5.1 Components

All components required to build and link a user application with the Solarflare TCPDirect API are distributed with Onload. When Onload is installed all required directories/files are located under the Onload distribution directory.

5.2 Compiling and Linking

5.2.1 Header files

Applications or libraries using TCPDirect include the `zf.h` header which is installed into the system include directory. For example:

```
#include <zf/zf.h>
```

5.2.2 Linking

The application will need to be linked either:

- with `libonload_zf_static.a` and `libciul1.a`, to link statically, or
- with `libonload_zf.so`, to link dynamically.

All of the above libraries are deployed to the system library directory by `onload_install`.

If compiling your application against TCPDirect libraries distributed with one version of Onload, and running on a system with a different version of Onload, some care is required. TCPDirect currently preserves compatibility and provides a stable API between the user-space components and the kernel drivers, so that applications compiled against an older TCPDirect library will work when run with newer drivers. Compatibility in the other direction (newer TCPDirect libraries running with older drivers) is not guaranteed. Finally, TCPDirect does not support linking dynamically against one version of the libraries, and then running against another.

The simplest approach is to link statically to `libonload_zf_static.a`, as this ensures that the version of the library used will match the one you have compiled against.

For those wishing to use TCPDirect in combination with Onload, it is possible to link either statically or dynamically to TCPDirect and then to run the application with the `onload` wrapper in the usual way to allow the Onload intercepts to take effect.

5.2.3 Debugging

By default, the TCPDirect libraries are optimized for performance, and in particular perform only a minimum of logging and parameter-validation. To aid testing, debug versions of the TCPDirect libraries are provided, which *do* offer such validation and logging. As with the production libraries, these are available both as static and as shared libraries.

To use the static debug library, an application must be linked against it explicitly, rather than being linked against the production library. The debug library is not installed to the linker's default search path, and so the full path to the library must be passed to the linker. The debug library is named `libonload_zf_static.a`, as is the production library, but is installed to the `zf/debug` subdirectory of the system library directory (typically `/usr/lib64`).

To use the shared debug library, the application should link as normal against the shared library as described in the Linking section above, but when run should be invoked via the `zf_debug` wrapper. For example, an application called `app` linked against the shared TCPDirect library will use the production library when invoked as

```
app
```

and will use the debug library when invoked as

```
zf_debug app
```

By default, the debug libraries emit the same logging messages as do the production libraries, but the `log_level` attribute described in the Attributes chapter can be used to enable additional logging selectively. As a convenience, the `-l` option to the `zf_debug` wrapper will set this attribute to the specified value.

5.3 General

The majority of the functions in this API will return 0 on success, or a negative error code on failure. These are negated values of standard Linux error codes as defined in the system's `errno.h`. `errno` itself is not used.

Most of the API is non-blocking. The cases where this is not the case (e.g. zf_muxer_wait()) are highlighted in the rest of this document.

The public API is defined by the headers in the `zf` subdirectory of the system include directory (typically `/usr/include`).

Attributes (defined by struct zf_attr) are used to pass configuration details through the API. This is similar to the existing SolarCapture attribute system.

The following sections discuss the most common operations. Zocket shutdown, obtaining addresses, and some other details are generally omitted for clarity – please refer to the suggested headers and example code for full details.

5.4 Using stacks

Before zockets can be created, the calling application must first create a stack using the following functions:

```
int zf_stack_alloc(struct zf_attr* attr, struct zf_stack** stack_out);
int zf_stack_free(struct zf_stack* stack);
```

The `attr` parameter to zf_stack_alloc() configures various aspects of the stack's behavior. In particular, the `interface` attribute specifies which network interface the stack should use, and the `n_bufs` attribute determines the total number of packet buffers allocated by the stack. Packet buffers are required to send and receive packets to and from the network, and also to queue packets on zockets for sending and receiving. A value of `n_bufs` that is too small can result in dropped packets and in various API calls failing with `ENOMEM`. Please see the Attributes chapter and the documentation for each API call for more details.

5.5 Using zockets

TCPDirect supports both TCP and UDP, but in contrast to the BSD sockets API the type of these zockets is explicit through the API types and function calls and UDP zockets are separated into receive (RX) and transmit (TX) parts.

5.6 UDP receive

First allocate a UDP receive zocket:

```
int zfur_alloc(struct zfur** us_out,
               struct zf_stack* st,
               const struct zf_attr* attr);
```

Then bind to associate the zocket with an address, port, and add filters:

```
int zfur_addr_bind(struct zfur* us,
                   struct sockaddr* laddr,
                   socklen_t laddrlen,
                   const struct sockaddr* raddr,
                   socklen_t raddrlen,
                   int flags);
```

Then receive packets:

```
int zfur_zc_recv(struct zfur* us,
                 struct zfur_msg* msg,
                 int flags);
```

zfur_zc_recv() will perform a zero-copy read of a single UDP datagram. The struct zfur_msg is completed to point to the buffers used by this message. Because it is zero-copy, the buffers used are locked (preventing re-use by the stack) until zfur_zc_recv_done() is called:

```
int zfur_zc_recv_done(struct zfur* us,
                      struct zfur_msg* msg);
```

Note

These functions can all be found in zf_udp.h.

5.7 UDP send

First allocate a UDP TX zocket, using the supplied addresses and ports:

```
int zfut_alloc(struct zfut** us_out,
               struct zf_stack* st,
               const struct sockaddr* laddr,
               socklen_t laddrlen,
               const struct sockaddr* raddr,
               socklen_t raddrlen,
               int flags,
               const struct zf_attr* attr);
```

Then perform a copy-based send (potentially using PIO) of a single datagram:

```
int zfut_send(struct zfut* us,
              const struct iovec* iov,
              int iov_cnt,
              int flags);
```

Note

These functions can all be found in zf_udp.h.

5.8 TCP listening

A TCP listening zocket can be created:

```
int zftl_listen(struct zf_stack* st,
                const struct sockaddr* laddr,
                socklen_t laddrlen,
                const struct zf_attr* attr,
                struct zftl** tl_out);
```

And a passively opened zocket accepted:

```
int zftl_accept(struct zftl* tl,
                struct zft** ts_out);
```

Listening zockets can be closed and freed:

```
int zftl_free(struct zftl* ts);
```

Note

These functions can all be found in zf_tcp.h.

5.9 TCP send and receive

Allocate a TCP (non-listening) zocket. Unlike UDP, this can be used for both send and receive:

```
int zft_alloc(struct zf_stack* st,
              const struct zf_attr* attr,
              struct zft_handle** handle_out);
```

Bind the zocket to a local address/port:

```
int zft_addr_bind(struct zft_handle* handle,
                  const struct sockaddr* laddr,
                  socklen_t laddrlen,
                  int flags);
```

Then connect the zocket to a remote address/port. Note that the supplied zocket handle is replaced with a different type as part of this operation. This function does not block (subsequent operations will return an error until it has completed).

```
int zft_connect(struct zft_handle* handle,
                const struct sockaddr* raddr,
                socklen_t raddrlen,
                struct zft** ts_out);
```

Perform a zero-copy receive on the connected TCP zocket:

```
int zft_zc_recv(struct zft* ts,
                struct zft_msg* msg,
                int flags);
```

The struct zft_msg is completed to point to the received message. Because it is zero-copy, this will lock the buffers used until the caller indicates that it has finished with them by calling:

```
void zft_zc_recv_done(struct zft* ts,
                      struct zft_msg* msg);
```

Alternatively a copy-based receive call can be made:

```
int zft_recv(struct zft* ts,
             struct iovec* iov_out,
             int iovcnt,
             int flags);
```

A copy-based send call can be made, and the supplied buffers reused immediately after this call returns:

```
int zft_send(struct zft* ts,
             const struct iovec* iov,
             int iov_cnt,
             int flags);
```

Note

These functions can all be found in zf_tcp.h.

5.10 Alternative Tx queues

Finally, for lowest latency on the fast path, a special API based around different alternative queues of data can be used. The TX alternative API is used to minimise latency on send, by pushing packets though the TX path on the NIC before a decision can be made whether they are needed.

```
int zf_alternatives_alloc(struct zf_stack* stack,
                          const struct zf_attr* attr,
                          zf_althandle* alt_out);
int zf_alternatives_release(struct zf_stack* stack,
                            zf_althandle alt);
int zf_alternatives_send(struct zf_stack* stack,
                         zf_althandle alt);
int zf_alternatives_cancel(struct zf_stack* stack,
                           zf_althandle alt);
int zft_alternatives_queue(struct zft* ts,
                           zf_althandle alt,
                           const struct iovec* iov,
                           int iov_cnt,
                           int flags);
unsigned zf_alternatives_free_space(struct zf_stack* stack,
                                    zf_althandle alt);
```

At the point when the decision to send is made the packet has already nearly reached the wire, minimising latency on the critical path.

Multiple queues are available for this, allowing alternative packets to be queued. Then when it is known what needs to be sent the appropriate alternative queue is selected. Packets queued on this are then sent to the wire.

When a packet is queued a handle is provided to allow future updates to the packet data. However, packet data update requires requeuing all packets on the affected alternative, so incurs a time penalty.

Here is an example, where there are 2 things that need updates, A and B, but it's not yet known which will be needed. The application has allocated 3 alternative queues, allowing them to queue updates for either A only, B only, or both:

```
zf_alternatives_alloc(ts, attr, &queue_a);
zft_alternatives_queue(ts, queue_a, <UpdateA_data>, flags);
zf_alternatives_alloc(ts, attr, &queue_b);
zft_alternatives_queue(ts, queue_b, <UpdateB_data>, flags);
zf_alternatives_alloc(ts, attr, &queue_ab);
zft_alternatives_queue(ts, queue_ab, <UpdateA_data>, flags);
zft_alternatives_queue(ts, queue_ab, <UpdateB_data>, flags);
```

After running the above code, the queues are as follows:

- queue_a: <UpdateA_data>
- queue_b: <UpdateB_data>
- queue_ab: <UpdateA_data><UpdateB_data>

A single packet can only be queued on one alternative. In the example above each instance of an update is a separate buffer.

When it is known which update is required the application can select the appropriate alternative. The zf_alternatives_send() function is used to do this. This will send out the packets on the selected alternative. If other alternatives have queued packets, you must flush them without sending them, as the TCP headers will then be incorrect on these packets. The zf_alternatives_cancel() function is used to do this.


```
zf_alternatives_send(ts, queue_a);
zf_alternatives_cancel(ts, queue_b);
zf_alternatives_cancel(ts, queue_ab);
```

After running the above code, the packet containing <UpdateA_data> has been sent from queue_a, and all three queues are empty and available for re-use.

Packet data cannot be edited in place once a packet has been queued on an alternative. If a queued packet needs to be updated it must be requeued, together with all other packets currently queued on the alternative. The zf_alternatives_cancel() and zft_alternatives_queue() functions are used to do this.

To avoid having to wait for the original alternative to be canceled before re-use a replacement alternative can be supplied. The unwanted alternative could then be freed:

```
zf_alternatives_alloc(ts, attr, &queue_new_ab);
zft_alternatives_queue(ts, queue_new_ab, <UpdateA_edited_data>, flags);
zft_alternatives_queue(ts, queue_new_ab, <UpdateB_data>, flags);
zf_alternatives_release(ts, queue_ab);
```

Before running the above code, queue_ab contains unwanted data for editing:

- queue_ab: <UpdateA_data><UpdateB_data>

After running the above code, queue_new_ab contains the new edited data, and queue_ab has been freed:

- queue_new_ab: <UpdateA_edited_data><UpdateB_data>

To determine the maximum packet size you can queue on an alternative, use the zf_alternatives_free_space() function.

```
fs = zf_alternatives_free_space(ts, queue_ab);
```

Note

These functions can all be found in zf_alts.h.

5.11 Epoll – muxer.h

The multiplexer allows multiple zockets to be polled in a single operation. The multiplexer owes much of its design (and some of its datatypes) to epoll.

The basic unit of functionality is the multiplexer set implemented by zf_muxer_set. Each type of zocket (e.g. UDP receive, UDP transmit, TCP listening, TCP) that can be multiplexed is equipped with a method for obtaining a zf_waitable that represents a given zocket:

```
struct zf_waitable* zfur_to_waitable(struct zfur* us);
struct zf_waitable* zfut_to_waitable(struct zfut* us);
struct zf_waitable* zftl_to_waitable(struct zftl* tl);
struct zf_waitable* zft_to_waitable(struct zft* ts);
```


This zf_waitable can then be added to a multiplexer set by calling zf_muxer_add(). Each waitable can only exist in a single multiplexer set at once. Each multiplexer set can only contain waitables from a single stack.

```
int zf_muxer_add(struct zf_muxer_set*,
                 struct zf_waitable* w,
                 const struct epoll_event* event);
```

Having added all of the desired zockets to a set, the set can be polled using zf_muxer_wait().

```
int zf_muxer_wait(struct zf_muxer_set*,
                  struct epoll_event* events,
                  int maxevents,
                  int64_t timeout);
```

This function polls a multiplexer set and populates an array of event descriptors representing the zockets in that set that are ready. The events member of each descriptor specifies the events for which the zocket is actually ready, and the data member is set to the user-data associated with that descriptor, as specified in the call to zf_muxer_add() or zf_muxer_mod().

Before checking for ready zockets, the function calls zf_reactor_perform() on the set's stack in order to process events from the hardware. In contrast to the rest of the API, zf_muxer_wait() can block. The maximum time to block is specified timeout, and a value of zero results in non-blocking behaviour. A negative value for timeout will allow the function to block indefinitely. If the function, blocks, it will call zf_reactor_perform() repeatedly in a tight loop.

The multiplexer supports only edge-triggered events: that is, if zf_muxer_wait() reports that an zocket is ready, it will not do so again until a new event occurs on that zocket, even if the zocket is in fact ready.

Waitables already in a set can be modified:

```
int zf_muxer_mod(struct zf_waitable* w,
                 const struct epoll_event* event);
```

and deleted from the set:

```
int zf_muxer_del(struct zf_waitable* w);
```

These functions can all be found in muxer.h.

5.12 Stack polling

The majority of the calls in the API are non-blocking and for performance reasons do not attempt to speculatively process events on a stack. The API provides the following function to allow the calling application to request the stack process events. It will return zero if nothing user-visible occurred as a result, or greater than zero if something potentially user-visible happened (e.g. received packet delivered to a zocket, zocket became writeable, etc). It may return false positives, i.e. report that something user-visible occurred, when in fact it did not.

```
int zf_reactor_perform(struct zf_stack* st);
```

Any calls which block (e.g. zf_muxer_wait()) will make this call internally. The code examples at the end of this document show how zf_reactor_perform() can be used.

The API also provides the following function to determine whether a stack has work pending. It will return non-zero if the stack has work pending, and therefore the application should call zf_reactor_perform() or zf_muxer_wait().

```
int zf_stack_has_pending_work(const struct zf_stack* st);
```

These functions can all be found in zf_reactor.h.

5.13 Miscellaneous

For TCP zockets you can discover the local and/or remote IP addresses and ports in use:

```
void zft_getname(struct zft* ts, struct sockaddr* laddr_out,
                 struct sockaddr* raddr_out);
```

These functions can all be found in zf_tcp.h and zf_udp.h.

5.14 zf_stackdump

TCPDirect does not use the Onload bypass datapaths, it uses its own datapath therefore traffic sent/received by TCPDirect stacks and zockets is NOT visible using tcpdump or onload_tcpdump or onload_stackdump.

The TCPDirect zf_stackdump feature can be used to analyse stacks/zockets created by the TCPDirect application.

5.14.1 Usage

```
# zf_stackdump -h
zf_stackdump [command [stack_ids...]]

Commands:
  list        List stack(s)
  dump        Show state of stack(s)

The default command is 'list'.  Commands iterate over all
stacks if no stacks are specified on the command line.
enp4s0f0/0f0         id=10    pid=8845
```

```
# zf_stackdump dump
============================================================
name=enp4s0f0/0f0
  pool: pkt_bufs_n=17536 free=17025
  config: tcp_timewait_ticks=666 tcp_finwait_ticks=666
  config: tcp_initial_cwnd=0 ms_per_tcp_tick=90
  alts: n_alts=0
  stats: ring_refill_nomem=0
nic0: vi=240 flags=0 intf=enp4s0f0 index=6 hw=1A1
  txq: pio_buf_size=2048
============================================================
UDP RX enp4s0f0/0f0:0
  filter: lcl=172.16.130.252:8012 rmt=0.0.0.0:0
  rx: unread=1 begin=0 process=0 end=1
  udp rx: release_n=1 q_drops=0
```

5.14.2 stackdump output: stack

Parameter	Description
pkt_bufs_n	num of packet buffers allocated to the stack.
free	num of free (available) packet buffers.
tcp_timewait_ticks	length of the TIME-WAIT timer in ticks.

tcp_finwait_ticks	length of the FIN-WAIT-2 timer in ticks.
tcp_initial_cwnd	size of TCP congestion window.
ms_per_tcp_tick	granularity of TCP timer in milliseconds.
n_alts	total number of TX alternatives allocated to this stack.
ring_refill_nomem	num times there were no free packet buffers to refill rx ring (increase buffers with n_bufs attr).
vi	VI being used by this stack.
flags	stack flags.
intf	physical interface being used.
index	index of the physical interface.
pio_buf_size	size (bytes) of a PIO buffer.

5.14.3 stackdump output: UDP RX

Parameter	Description
filter	identifies filters installed on the adapter.
rx unread	num packets received, but still in zocket buffer rx queue
rx begin	zf_rx_ring: oldest pkt buffer not yet read.
rx process	zf_rx_ring: oldest pkt buffer not yet processed - so buffer not yet reaped.
rx end	zf_rx_ring: index of the last pkt in the queue.
release_n	num zero-copy packet awaiting release.
q_drops	num packets dropped from the zockets rx queue.

5.14.4 stackdump output: UDP TX

Parameter	Description
UDP TX	local interface local_ip:port remote_ip:port.
path	dst server src server.
tx posted	num tx descriptors posted to the NIC.
tx completed	num tx descriptors that have completed TX.

5.14.5 stackdump output: TCP TX/RX

Parameter	Description
tx posted	num tx descriptors posted to the NIC.
tx completed	num tx descriptors that have completed TX.
rx	unread, begin, process end, see UDP RX.
flags	zocket flags.
flags_ack_delay	ACK flags TF_ACK_DELAY 0x01 | TF_ACK_NOW 0x02 | TF_INTR 0x04 (in fast recovery) | TF_ACK_NEXT 0x08.
parent	identifies the listeing zocket from which a passive-open zocket was accepted.

refcount	when a zocket is used both from the TCP state machine and the application. This allows us to track when both have finished using it, and it can be freed.
snd_nxt	next sequence num to send.
lastack	last acknowledged sequence num.
snd_wnd_max	size of size window advertised by the peer.
snd_wl1	max sequence number advertised.
snd_wl2	last sequence number acknowledged.
snd_lbb	sequence num of next byte to be buffered.
snd_right_edge	sequence num of TCP send window.
send	num segments held in the send buffer.
inflight	num segments sent, but not yet acknowledged.
qbegin	TCP segment at sendq start.
qmiddle	TCP seqment at sendq middle.
qend	TCP segment at sendq end.
sndbuf	zocket send buffer size (bytes).
cwnd	size of congestion avoidance window.
ssthresh	slow start threshold - num bytes that have to be sent before exiting slow start.
mss_lim	max segment size limit set by peer.
rcv_nxt	next expected sequence number.
rcv_ann_wnd	receiver window to announce.
rcv_ann_right_edge	announced right edge of window.
mss	max segment size.
seq	sequence number used for RTT estimation.
sa	rtt variable: smoothed round trip time.
sv	rtt variable: round trip time variance estimate.
nrtx	num of RTO retransmission attempts - reset to zero when a new ACK is recveived.
dupacks	num duplicate acks received.
persist_backoff	num of zero send win probes - sends probe pkt to keep connection alive.
timers	active timers.
ooo: added	out of order pkt added to sendq.
ooo: removed	count removals from overflow including segments that become in-order.
ooo: replaced	out of order pkt replaced in sendq.
ooo: handling deferred	count of deffered out-of-order pkts.
ooo: dropped_nomem	num of out of order pkts dropped when memory allocation fails.
ooo: drop_overfilled	num of out of order pkts dropped to prevent buffer overflowing.
msg_more_send_delayed	num of times there was no send because of MSG_MORE flag.
send_nomem	num of times there were no free packet buffers to perform a send.

Chapter 6

Worked Examples

This part of the documentation examines simplified versions of zfudppingpong and zftcppingpong. These are small applications which listen for packets and replies, with as low latency as possible.

Note

These examples do not set the values of attributes programmatically. Instead, they are left with the values set from the defaults and the `ZF_ATTR` environment variable. In particular, it is necessary to set the value of the `interface` attribute in `ZF_ATTR` in order to use these examples. Fully-fledged applications might prefer instead to set attributes using (for example) zf_attr_set_str(). Please see the Attributes chapter and the documentation for attr.h for more information.

6.1 UDP ping pong example

In the following example various boiler plate code has been omitted for clarity. For a full usable example see `src/tests/zf_apps/zfudppingpong.c`.

```
void ping(struct zf_stack* stack, struct zfut* ut, struct zfur* ur)
{
  unsigned char data[1500];
  struct iovec siov = { data, 1};
  struct {
    struct zfur_msg zcr;
    struct iovec iov[2];
  } rd = { { .iovcnt = 2 } };

  uint64_t* ping_last_word = cfg.size>=8 ? ((uint64_t*)&data[cfg.size]) -1 : 0;

  siov.iov_len = cfg.size;
  if( ping_last_word )
    *ping_last_word = 0x1122334455667788;
  ZF_TRY(zfut_send(ut, &siov, 1, 0));

  for(int it = 0; it < cfg.itercount;) {
    while(zf_reactor_perform(stack) == 0);

    rd.zcr.iovcnt = 2;
    ZF_TRY(zfur_zc_recv(ur, &rd.zcr, 0));

    if( rd.zcr.iovcnt == 0 )
      continue;

    it += rd.zcr.iovcnt;

    zfur_zc_recv_done(ur, &rd.zcr);
```



```
    if( ping_last_word )
      ++*ping_last_word;
    ZF_TRY(zfut_send(ut, &siov, 1, 0));
  }
}


void pong(struct zf_stack* stack, struct zfut* ut, struct zfur* ur)
{
  struct {
    struct zfur_msg zcr;
    struct iovec iov[2];
  } rd = { { .iovcnt = 2 } };

  for(int it = 0; it < cfg.itercount;) {
    while(zf_reactor_perform(stack) == 0);

    rd.zcr.iovcnt = 2;
    ZF_TRY(zfur_zc_recv(ur, &rd.zcr, 0));

    if( rd.zcr.iovcnt == 0 )
      continue;

    /* in pong we reply with the same data */
    for( int i = 0 ; i < rd.zcr.iovcnt; ++i ) {
      ZF_TRY(zfut_send(ut, &rd.zcr.iov[i], 1, 0));
    }

    it += rd.zcr.iovcnt;
    zfur_zc_recv_done(ur, &rd.zcr);
  }
}


int main(int argc, char* argv[])
{
  ZF_TRY(zf_init());

  struct zf_attr* attr;
  ZF_TRY(zf_attr_alloc(&attr));

  struct zf_stack* stack;
  ZF_TRY(zf_stack_alloc(attr, &stack));

  struct zfur* ur;
  ZF_TRY(zfur_alloc(&ur, stack, attr));

  struct sockaddr_in laddr = parse_addr((char*)cfg.laddr);
  struct sockaddr_in raddr = parse_addr((char*)cfg.raddr);
  ZF_TRY(zfur_addr_bind(ur, &laddr, &raddr, 0));

  struct zfut* ut;
  ZF_TRY(zfut_alloc(&ut, stack, &laddr, &raddr, 0, attr));

  (cfg.ping ? &ping : pong)(stack, ut, ur);

  return 0;
}
```

6.2 TCP pong example

In the following example some boiler plate code has been omitted for clarity. For a full usable example see `src/tests/zf_apps/zftcppingpong.c`, which includes the “ping” side, how to use the multiplexer, and other details.

```
int main(int argc, const char* argv[])
{
  ZF_TRY(zf_init());

  struct zf_attr* attr;
  ZF_TRY(zf_attr_alloc(&attr));

  struct zf_stack* stack;
  ZF_TRY(zf_stack_alloc(attr, &stack));
```

```
  struct zft_handle* tcp_handle;
  ZF_TRY(zft_alloc(stack, attr, &tcp_handle));

  struct sockaddr_in laddr = {
    .sin_addr = { inet_addr(argv[1]) },
    .sin_port = htons(2000),
  };
  struct sockaddr_in raddr = {
    .sin_addr = { inet_addr(argv[2]) },
    .sin_port = htons(2000),
  };
  ZF_TRY(zft_addr_bind(tcp_handle, &laddr));

  struct zft* tcp;
  ZF_TRY(zft_connect(tcp_handle, &raddr, &tcp));

  int first_recv = 1;
  unsigned char data[1500];
  struct iovec siov = { data, 1};
  struct {
    struct zft_msg zcr;
    struct iovec __iov_tail;
  } rd = { { .iovcnt = 1 } };

  while(1) {
    while(zf_reactor_perform(stack) == 0);

    rd.zcr.iovcnt = 1;
    ZF_TRY(zft_zc_recv(tcp, &rd.zcr, 0));

    if( rd.zcr.iovcnt == 0 )
      continue;

    if( first_recv ) {
      first_recv = 0;

      siov.iov_len = rd.zcr.iov[0].iov_len;
      memcpy(data, ((char*)rd.zcr.iov[0].iov_base), siov.iov_len);
    }
    for( int i = 0 ; i < rd.zcr.iovcnt; ++i ) {
      ZF_TRY(zft_send(tcp, &siov, 1, 0));
    }
    zft_zc_recv_done(tcp, &rd.zcr);
 }

  ZF_TRY(zft_shutdown_tx(tcp));
  ZF_TRY(zft_free(tcp));

  return 0;
}
```


Chapter 7

Attributes

Many TCPDirect API functions take an *attribute object* of type zf_attr. Each attribute object specifies a set of attributes, which are key-value pairs. These attributes are documented in this section.

Attribute	Description
alt_buf_size	Amount of NIC-side buffer space to allocate for use with TCP alternatives on this VI.
alt_count	Number of TCP alternatives to allocate on this VI.
arp_reply_timeout	Maximum time to wait for ARP replies, in microseconds (approx).
interface	Use this interface name as zf_stack interface.
log_format	Combination of flags: ZF_LF_STACK_NAME (0x1), ZF_LF_FRC(0x2), ZF_LF_TCP_TIME(0x4), ZF_LF_PROCESS(0x8)
log_level	Bitmask to enable different log message levels for each logging component.
max_tcp_endpoints	Sets the maximum number of TCP endpoints (i.e. struct zft).
max_tcp_listen_endpoints	Sets the maximum number of TCP endpoints (i.e. struct zftl).
max_tcp_syn_backlog	Sets the maximum number of half-open connections maintained in the stack.
max_udp_rx_endpoints	Sets the maximum number of UDP RX endpoints (i.e. struct zfur).
max_udp_tx_endpoints	Sets the maximum number of UDP TX endpoints (i.e. struct zfut).
n_bufs	Number of packet buffers to allocate for the stack.
name	The object name.
reactor_spin_count	Sets how many iterations of the event processing loop zf_reactor_perform() will make (in the absence of any events) before returning.
rx_ring_max	Set the size and maximum fill level of the RX descriptor ring, which provides buffering between the network adapter and software.

rx_ring_refill_batch_size	Sets the number of packet buffers rx ring is refilled with on each zf_reactor_perform call.
rx_ring_refill_interval	Sets the frequency of rx buffer ring refilling during inner zf_reactor_perform() loop.
tcp_alt_ack_rewind	The maximum number of bytes by which outgoing ACKs will be allowed to go backwards when sending an alternative queue.
tcp_delayed_ack	Enable TCP delayed ACK ("on" by default).
tcp_finwait_ms	Length of TCP FIN-WAIT-2 timer in ms, 0 - disabled.
tcp_initial_cwnd	The initial congestion window for new TCP zockets.
tcp_retries	The maximum number of TCP retransmits if data is not acknowledged by the network peer in general case.
tcp_syn_retries	The maximum number of TCP SYN retransmits during zft_connect().
tcp_synack_retries	The maximum number of TCP SYN-ACK retransmits before incoming connection is dropped.
tcp_timewait_ms	Length of TCP TIME-WAIT timer in ms.
tcp_wait_for_time_wait	Do not consider a stack to be quiescent if there are any TCP zockets in the TIME_WAIT state.
tx_ring_max	Set the size of the TX descriptor ring, which provides buffering between the software and the network adaptor.

7.1 alt_buf_size Attribute Reference

Amount of NIC-side buffer space to allocate for use with TCP alternatives on this VI.

Detailed Description

Type

Integer.

Default

40960.

Relevant components

zf_vi.

7.2 alt_count Attribute Reference

Number of TCP alternatives to allocate on this VI.

Detailed Description

Type

Integer.

Default

0.

Relevant components

zf_vi.

7.3 arp_reply_timeout Attribute Reference

Maximum time to wait for ARP replies, in microseconds (approx).

Detailed Description

Type

Integer.

Default

1000.

Relevant components

zf_stack.

7.4 interface Attribute Reference

Use this interface name as zf_stack interface.

Detailed Description

Type

String.

Default

none.

Relevant components

zf_stack.

7.5 log_format Attribute Reference

Combination of flags: ZF_LF_STACK_NAME (0x1), ZF_LF_FRC(0x2), ZF_LF_TCP_TIME(0x4), ZF_LF_PROCESS(0x8)

Detailed Description

Type

Integer.

Default

stack name and tcp time.

Relevant components

zf_stack.

7.6 log_level Attribute Reference

Bitmask to enable different log message levels for each logging component.

Detailed Description

The log message level for each component is specified using a separate 4 bit nibble within the bitmask. The value of each nibble is a bitwise combination of: 0(none), 0x1(errors), 0x2(warnings), 0x4(info), 0x8(trace - debug build only). The following components are available: stack (bits 0-3), TCP-rx (4-7), TCP-tx (8-11), TCP-connection (12-15), UDP-rx (16-19), UDP-tx (20-23), UDP-connection (24-27), muxer (28-31), pool (32-35), fast-path (36-39), timers (40-43), filters (44-47), cplane (48-51). E.g. 0xfff0 will enable all TCP related logging and disable all other logging.

Type

Bitmask.

Default

ERR-level on all components.

Relevant components

zf_stack.

7.7 max_tcp_endpoints Attribute Reference

Sets the maximum number of TCP endpoints (i.e. struct zft).

Detailed Description

Type

Integer.

Default

64.

Relevant components

zf_stack.

7.8 max_tcp_listen_endpoints Attribute Reference

Sets the maximum number of TCP endpoints (i.e. struct zftl).

Detailed Description

Type

Integer.

Default

16.

Relevant components

zf_stack.

7.9 max_tcp_syn_backlog Attribute Reference

Sets the maximum number of half-open connections maintained in the stack.

Detailed Description

Type

Integer.

Default

net.ipv4.tcp_max_syn_backlog.

Relevant components

zf_stack.

7.10 max_udp_rx_endpoints Attribute Reference

Sets the maximum number of UDP RX endpoints (i.e. struct zfur).

Detailed Description

Type

Integer.

Default

64.

Relevant components

zf_stack.

7.11 max_udp_tx_endpoints Attribute Reference

Sets the maximum number of UDP TX endpoints (i.e. struct zfut).

Detailed Description

Type

Integer.

Default

64.

Relevant components

zf_stack.

7.12 n_bufs Attribute Reference

Number of packet buffers to allocate for the stack.

Detailed Description

The optimal value for this parameter depends on the size of the RX and TX queues, the total number of zockets in the stack, the number of alternatives in use and the frequency at which the application polls the stack and reads pending data from zockets. 0 - use maximum the stack with given parameters can use.

Type

Integer.

Default

0.

Relevant components

zf_pool.

7.13 name Attribute Reference

The object name.

Detailed Description

Object names are visible in log messages, but have no other effect.

Type

String.

Default

(none).

Relevant components

zf_stack, zf_pool, zf_vi.

7.14 reactor_spin_count Attribute Reference

Sets how many iterations of the event processing loop zf_reactor_perform() will make (in the absence of any events) before returning.

Detailed Description

The default value makes zf_reactor_perform() briefly spin if there are no new events present. A higher number can give better latency, however zf_reactor_perform() will take more time to return when no new events are present. The minimum value is 1, which disables spinning.This attribute also affects the cost of zf_muxer_wait() when invoked with timeout_ns=0.

Type

Integer.

Default

128.

Relevant components

zf_stack.

7.15 rx_ring_max Attribute Reference

Set the size and maximum fill level of the RX descriptor ring, which provides buffering between the network adapter and software.

Detailed Description

The RX ring sizes supported are 512, 1024, 2048 and 4096. The n_bufs attribute may need to be increased when changing this value.

Type

Integer.

Default

512.

Relevant components

zf_vi.

7.16 rx_ring_refill_batch_size Attribute Reference

Sets the number of packet buffers rx ring is refilled with on each zf_reactor_perform call.

Detailed Description

Must be multiple of 8.

Type

Integer.

Default

16.

Relevant components

zf_stack.

7.17 rx_ring_refill_interval Attribute Reference

Sets the frequency of rx buffer ring refilling during inner zf_reactor_perform() loop.

Detailed Description

Set to 1 to have the ring refilled at each iteration.

Type

Integer.

Default

1.

Relevant components

zf_stack.

7.18 tcp_alt_ack_rewind Attribute Reference

The maximum number of bytes by which outgoing ACKs will be allowed to go backwards when sending an alternative queue.

Detailed Description

Type

Integer.

Default

64K.

Relevant components

zf_stack.

7.19 tcp_delayed_ack Attribute Reference

Enable TCP delayed ACK ("on" by default).

Detailed Description

Type

Integer.

Default

1.

Relevant components

zf_stack.

7.20 tcp_finwait_ms Attribute Reference

Length of TCP FIN-WAIT-2 timer in ms, 0 - disabled.

Detailed Description

Type

Integer.

Default

net.ipv4.tcp_fin_timeout.

Relevant components

zf_stack.

7.21 tcp_initial_cwnd Attribute Reference

The initial congestion window for new TCP zockets.

Detailed Description

Type

Integer.

Default

10 ∗ MSS.

Relevant components

zf_stack.

7.22 tcp_retries Attribute Reference

The maximum number of TCP retransmits if data is not acknowledged by the network peer in general case.

Detailed Description

See also tcp_synack_retries, tcp_syn_retries.

Type

Integer.

Default

net.ipv4.tcp_retries2.

Relevant components

zf_stack.

7.23 tcp_syn_retries Attribute Reference

The maximum number of TCP SYN retransmits during zft_connect().

Detailed Description

Type

Integer.

Default

net.ipv4.tcp_syn_retries.

Relevant components

zf_stack.

7.24 tcp_synack_retries Attribute Reference

The maximum number of TCP SYN-ACK retransmits before incoming connection is dropped.

Detailed Description

Type

Integer.

Default

net.ipv4.tcp_synack_retries.

Relevant components

zf_stack.

7.25 tcp_timewait_ms Attribute Reference

Length of TCP TIME-WAIT timer in ms.

Detailed Description

Type

Integer.

Default

net.ipv4.tcp_fin_timeout.

Relevant components

zf_stack.

7.26 tcp_wait_for_time_wait Attribute Reference

Do not consider a stack to be quiescent if there are any TCP zockets in the TIME_WAIT state.

Detailed Description

("off" by default).

Type

Integer.

Default

0.

Relevant components

zf_stack.

7.27 tx_ring_max Attribute Reference

Set the size of the TX descriptor ring, which provides buffering between the software and the network adaptor.

Detailed Description

The requested value is rounded up to the next size supported by the adapter. At time of writing the ring sizes supported are 512, 1024 and 2048. The n_bufs attribute may need to be increased when changing this value.

Type

Integer.

Default

512.

Relevant components

zf_vi.

Chapter 8

Data Structure Index

8.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 9

File Index

9.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 10

Data Structure Documentation

10.1 zf_attr Struct Reference

Attribute object.

```
#include <zf/attr.h>
```

10.1.1 Detailed Description

Attribute object.

Attributes are used to specify optional behaviours and parameters, usually when allocating objects. Each attribute object defines a complete set of the attributes that the stack understands.

For example, the "max_udp_rx_endpoints" attribute controls how many UDP-receive zockets can be created per zf_stack.

The default values for attributes may be overridden by setting the environment variable ZF_ATTR. For example:

```
** ZF_ATTR="interface=enp4s0f0;log_level=3"
**
```

Each function that takes an attribute argument will only be interested in a subset of the attributes specified by an zf_attr instance. Other attributes are ignored.

The set of attributes supported may change between releases, so applications should where possible tolerate failures when setting attributes.

The documentation for this struct was generated from the following file:

- attr.h

10.2 zf_muxer_set Struct Reference

Multiplexer set.

```
#include <zf/muxer.h>
```

10.2.1 Detailed Description

Multiplexer set.

Represents multiple objects (including zockets) that can be polled simultaneously.

The documentation for this struct was generated from the following file:

- muxer.h

10.3 zf_stack Struct Reference

```
#include <zf/zf_stack.h>
```

10.3.1 Detailed Description

A stack encapsulates hardware and protocol state. It is the fundamental object used to drive TCPDirect. Individual objects for handling TCP and UDP traffic — *zockets* — are created within a stack.

See Also

zf_stack_alloc()
zf_stack_free()
zf_reactor_perform()

The documentation for this struct was generated from the following file:

- zf_stack.h

10.4 zf_waitable Struct Reference

Abstract multiplexable object.

```
#include <zf/muxer.h>
```


10.4.1 Detailed Description

Abstract multiplexable object.

Zockets that can be added to a multiplexer set can be represented by a pointer of this type, which can be obtained by making the appropriate API call for the given zocket.

A waitable can also be retrieved for a stack by calling zf_stack_to_waitable(). Such waitables indicate whether a stack has quiesced, in the sense documented at zf_stack_is_quiescent().

Definition at line 43 of file muxer.h.

The documentation for this struct was generated from the following file:

- muxer.h

10.5 zft Struct Reference

Opaque structure describing a TCP zocket that is connected.

```
#include <zf/zf_tcp.h>
```

10.5.1 Detailed Description

Opaque structure describing a TCP zocket that is connected.

Definition at line 142 of file zf_tcp.h.

The documentation for this struct was generated from the following file:

- zf_tcp.h

10.6 zft_handle Struct Reference

Opaque structure describing a TCP zocket that is passive and not connected.

```
#include <zf/zf_tcp.h>
```

10.6.1 Detailed Description

Opaque structure describing a TCP zocket that is passive and not connected.

The documentation for this struct was generated from the following file:

- zf_tcp.h

10.7 zft_msg Struct Reference

TCP zero-copy RX message structure.

```
#include <zf/zf_tcp.h>
```

Data Fields

- int reserved [4]
- int pkts_left
- int flags
- int iovcnt
- struct iovec iov [ZF_FLEXIBLE_ARRAY_COUNT]

10.7.1 Detailed Description

TCP zero-copy RX message structure.

This structure is passed to zft_zc_recv(), which will populate it with pointers to received packets.

Definition at line 379 of file zf_tcp.h.

10.7.2 Field Documentation

10.7.2.1 int flags

Reserved.

Definition at line 385 of file zf_tcp.h.

10.7.2.2 struct iovec iov[ZF_FLEXIBLE_ARRAY_COUNT]

In: base of iovec array; out: filled with iovecs pointing to the payload of the received packets.

Definition at line 391 of file zf_tcp.h.

10.7.2.3 int iovcnt

In: Length of iov array expressed as a count of iovecs; out: number of entries of iov populated with pointers to packets.

Definition at line 388 of file zf_tcp.h.

10.7.2.4 int pkts_left

Out: Number of outstanding packets in the queue after this read.

Definition at line 383 of file zf_tcp.h.

10.7.2.5 int reserved[4]

Reserved.

Definition at line 381 of file zf_tcp.h.

The documentation for this struct was generated from the following file:

- zf_tcp.h

10.8 zftl Struct Reference

Opaque structure describing a TCP listening zocket.

```
#include <zf/zf_tcp.h>
```

10.8.1 Detailed Description

Opaque structure describing a TCP listening zocket.

Definition at line 34 of file zf_tcp.h.

The documentation for this struct was generated from the following file:

- zf_tcp.h

10.9 zfur Struct Reference

Opaque structure describing a UDP-receive zocket.

```
#include <zf/zf_udp.h>
```

10.9.1 Detailed Description

Opaque structure describing a UDP-receive zocket.

Definition at line 24 of file zf_udp.h.

The documentation for this struct was generated from the following file:

- zf_udp.h

10.10 zfur_msg Struct Reference

UDP zero-copy RX message structure.

```
#include <zf/zf_udp.h>
```

Data Fields

- int reserved [4]
- int dgrams_left
- int flags
- int iovcnt
- struct iovec iov [ZF_FLEXIBLE_ARRAY_COUNT]

10.10.1 Detailed Description

UDP zero-copy RX message structure.

This structure is passed to zfur_zc_recv(), which will populate it with pointers to received packets.

Definition at line 121 of file zf_udp.h.

10.10.2 Field Documentation

10.10.2.1 int dgrams_left

Out: Number of outstanding datagrams in the queue after this read.

Definition at line 125 of file zf_udp.h.

10.10.2.2 int flags

Reserved.

Definition at line 127 of file zf_udp.h.

10.10.2.3 struct iovec iov[ZF_FLEXIBLE_ARRAY_COUNT]

In: base of iovec array; out: filled with iovecs pointing to the payload of the received packets.

Definition at line 133 of file zf_udp.h.

10.10.2.4 int iovcnt

In: Length of iov array expressed as a count of iovecs; out: number of entries of iov populated with pointers to packets.

Definition at line 130 of file zf_udp.h.

10.10.2.5 int reserved[4]

Reserved.

Definition at line 123 of file zf_udp.h.

The documentation for this struct was generated from the following file:

- zf_udp.h

10.11 zfut Struct Reference

Opaque structure describing a UDP-transmit zocket.

```
#include <zf/zf_udp.h>
```

10.11.1 Detailed Description

Opaque structure describing a UDP-transmit zocket.

A UDP-transmit zocket encapsulates the state required to send UDP datagrams. Each such zocket supports only a single destination address.

Definition at line 226 of file zf_udp.h.

The documentation for this struct was generated from the following file:

- zf_udp.h

Chapter 11

File Documentation

11.1 attr.h File Reference

TCPDirect API for attribute objects.

Functions

- int zf_attr_alloc (struct zf_attr ∗∗attr_out)

  Allocate an attribute object.
- void zf_attr_free (struct zf_attr ∗attr)

  Free an attribute object.
- void zf_attr_reset (struct zf_attr ∗attr)

  Return attributes to their default values.
- int zf_attr_set_int (struct zf_attr ∗attr, const char ∗name, int64_t val)

  Set an attribute to an integer value.
- int zf_attr_get_int (struct zf_attr ∗attr, const char ∗name, int64_t ∗val)

  Get an integer-valued attribute.
- int zf_attr_set_str (struct zf_attr ∗attr, const char ∗name, const char ∗val)

  Set an attribute to a string value.
- int zf_attr_get_str (struct zf_attr ∗attr, const char ∗name, char ∗∗val)

  Get a string-valued attribute.
- int zf_attr_set_from_str (struct zf_attr ∗attr, const char ∗name, const char ∗val)

  Set an attribute from a string value.
- int zf_attr_set_from_fmt (struct zf_attr ∗attr, const char ∗name, const char ∗fmt,...)

  Set an attribute to a string value (with formatting).
- struct zf_attr ∗ zf_attr_dup (const struct zf_attr ∗attr)

  Duplicate an attribute object.
- int zf_attr_doc (const char ∗attr_name_opt, const char ∗∗∗docs_out, int ∗docs_len_out)

  Returns documentation for an attribute.

11.1.1 Detailed Description

TCPDirect API for attribute objects.

Definition in file attr.h.

11.1.2 Function Documentation

11.1.2.1 int zf_attr_alloc (struct zf_attr ** *attr_out*)

Allocate an attribute object.

Parameters

attr_out	The attribute object is returned here.

Returns

0 on success, or a negative error code:
-ENOMEM if memory could not be allocated
-EINVAL if the ZF_ATTR environment variable is malformed.

11.1.2.2 int zf_attr_doc (const char * *attr_name_opt,* const char *** *docs_out,* int * *docs_len_out*)

Returns documentation for an attribute.

Parameters

attr_name_opt	The attribute name.
docs_out	On success, the resulting doc string output.
docs_len_out	On success, the length of the doc string output.

Returns

0 on success, or a negative error code.

11.1.2.3 struct zf_attr* zf_attr_dup (const struct zf_attr * *attr*)

Duplicate an attribute object.

Parameters

attr	The attribute object.

Returns

A new attribute object.

This function is useful when you want to make non-destructive changes to an existing attribute object.

11.1.2.4 void zf_attr_free (struct zf_attr * *attr*)

Free an attribute object.

Parameters

attr	The attribute object.

11.1.2.5 int zf_attr_get_int (struct zf_attr * *attr*, const char * *name*, int64_t * *val*)

Get an integer-valued attribute.

Parameters

attr	The attribute object.
name	Name of the attribute.
val	Value of the attribute (output).

Returns

0 on success, or a negative error code: -ENOENT if `name` is not a valid attribute name -EINVAL if `name` does not have an integer type

11.1.2.6 int zf_attr_get_str (struct zf_attr * *attr*, const char * *name*, char ** *val*)

Get a string-valued attribute.

Parameters

attr	The attribute object.
name	Name of the attribute.
val	Value of the attribute (output). This is allocated with strdup() and must be free()ed by the caller.

Returns

0 on success, or a negative error code: -ENOENT if `name` is not a valid attribute name -EINVAL if `name` does not have a string type

11.1.2.7 void zf_attr_reset (struct zf_attr * *attr*)

Return attributes to their default values.

Parameters

attr	The attribute object.

11.1.2.8 int zf_attr_set_from_fmt (struct zf_attr * *attr*, const char * *name*, const char * *fmt*, ...)

Set an attribute to a string value (with formatting).

Parameters

attr	The attribute object.
name	Name of the attribute.
fmt	Format string for the new attribute value.

Returns

0 on success, or a negative error code:
-ENOENT if `name` is not a valid attribute name
-EINVAL if it is not possible to convert `fmt` to a valid value for the attribute
-EOVERFLOW if `fmt` is not within the range of values this attribut can take.

This function behaves exactly as zf_attr_set_from_str(), except that the string value is generated from a printf()-style format string.

11.1.2.9 int zf_attr_set_from_str (struct zf_attr ∗ *attr*, const char ∗ *name*, const char ∗ *val*)

Set an attribute from a string value.

Parameters

attr	The attribute object.
name	Name of the attribute.
val	New value for the attribute.

Returns

0 on success, or a negative error code:
-ENOENT if `name` is not a valid attribute name
-EINVAL if it is not possible to convert `val` to a valid value for the attribute
-EOVERFLOW if `val` is not within the range of values this attribut can take.

11.1.2.10 int zf_attr_set_int (struct zf_attr ∗ *attr*, const char ∗ *name*, int64_t *val*)

Set an attribute to an integer value.

Parameters

attr	The attribute object.
name	Name of the attribute.
val	New value for the attribute.

Returns

0 on success, or a negative error code:
-ENOENT if `name` is not a valid attribute name
-EOVERFLOW if `val` is not within the range of values this attribute can take.

11.1.2.11 int zf_attr_set_str (struct zf_attr ∗ *attr*, const char ∗ *name*, const char ∗ *val*)

Set an attribute to a string value.

Parameters

attr	The attribute object.
name	Name of the attribute.
val	New value for the attribute (may be NULL).

Returns

0 on success, or a negative error code:
-ENOENT if `name` is not a valid attribute name
-ENOMSG if the attribute is not a string attribute.

11.2 muxer.h File Reference

TCPDirect multiplexer.

```
#include <sys/epoll.h>
```

Functions

- int zf_muxer_alloc (struct zf_stack ∗stack, struct zf_muxer_set ∗∗muxer_out)
  Allocates a multiplexer set.
- void zf_muxer_free (struct zf_muxer_set ∗muxer)
  Frees a multiplexer set.
- int zf_muxer_add (struct zf_muxer_set ∗muxer, struct zf_waitable ∗w, const struct epoll_event ∗event)
  Adds a waitable object to a multiplexer set.
- int zf_muxer_mod (struct zf_waitable ∗w, const struct epoll_event ∗event)
  Modifies the event data for a waitable object in a multiplexer set.
- int zf_muxer_del (struct zf_waitable ∗w)
  Removes a waitable object from a multiplexer set.
- int zf_muxer_wait (struct zf_muxer_set ∗muxer, struct epoll_event ∗events, int maxevents, int64_t timeout_ns)
  Polls a multiplexer set.
- struct epoll_event ∗ zf_waitable_event (struct zf_waitable ∗w)
  Find out the epoll_event data in use with this waitable.
- int zf_waitable_fd_get (struct zf_stack ∗stack, int ∗fd)
  Create an fd that can be used within an epoll set or other standard muxer.
- int zf_waitable_fd_prime (struct zf_stack ∗stack)
  Prime the fd before blocking.

11.2.1 Detailed Description

TCPDirect multiplexer. The multiplexer, which allows multiple objects to be polled in a single operation.

The multiplexer allows multiple zockets to be polled in a single operation. The basic unit of functionality is the *multiplexer set* implemented by zf_muxer_set. Each type of zocket that can be multiplexed is equipped with a method for obtaining a zf_waitable that represents a given zocket; this zf_waitable can then be added to a multiplexer set by calling zf_muxer_add(). Having added all of the desired zockets to a set, the set can be polled using zf_muxer_-wait().

The multiplexer owes much of its design (and some of its datatypes) to `epoll(7)`.

Definition in file muxer.h.

11.2.2 Function Documentation

11.2.2.1 int zf_muxer_add (struct zf_muxer_set * *muxer*, struct zf_waitable * *w*, const struct epoll_event * *event*)

Adds a waitable object to a multiplexer set.

Parameters

muxer	Multiplexer set.
w	Waitable to add.
event	Descriptor specifying the events that will be polled on the waitable, and the data to be returned when those events are detected.

Returns

0 on success, or a negative error code:
-EXDEV Waitable does not belong to the multiplexer set's stack.
-EALREADY Waitable is already in this multiplexer set.
-EBUSY Waitable is already in another multiplexer set.

Adds a waitable object to a multiplexer set. Each waitable may belong to at most one multiplexer set at a time. The events of interest are specified by `event.events`, which is a bitfield that should be populated from one or more of `EPOLLIN`, `EPOLLOUT`, `EPOLLHUP` and `EPOLLERR` as desired. `event.data` specifies the data to be returned to a caller of zf_muxer_wait() when that waitable is ready. Note that the waitable itself is not in general returned to such callers; if this is desired, then `event.data` must be set in such a way that the waitable can be determined.

Note

Unlike epoll functions in Linux, you have to explicitly set `EPOLLHUP` and `EPOLLERR` if you want to be notified about these events.

11.2.2.2 int zf_muxer_alloc (struct zf_stack * *stack*, struct zf_muxer_set ** *muxer_out*)

Allocates a multiplexer set.

Parameters

stack	Stack to associate with multiplexer set.
muxer_out	Holds the address of the allocated multiplexer set on success.

Returns

0 on success, or a negative error code:
-ENOMEM Out of memory.

Allocates a multiplexer set, which allows multiple waitable objects to be polled in a single operation. Waitable objects, together with a mask of desired events, can be added to the set using zf_muxer_add(). The set can then be polled using zf_muxer_wait().

11.2.2.3 int zf_muxer_del (struct zf_waitable * *w*)

Removes a waitable object from a multiplexer set.

Parameters

w	Waitable to remove.

Returns

0 on success, or a negative error code:
-EINVAL `w` has not been added to a multiplexer set.

Note

This operation should be avoided on fast paths.

11.2.2.4 void zf_muxer_free (struct zf_muxer_set * *muxer*)

Frees a multiplexer set.

Parameters

muxer	The multiplexer set to free.

Note

If there are waitables in the set at the point at which it is freed, the underlying memory will not be freed until all of those waitables have been removed from the set. Nonetheless, the caller should never continue to use a pointer passed to this function.

11.2.2.5 int zf_muxer_mod (struct zf_waitable * *w*, const struct epoll_event * *event*)

Modifies the event data for a waitable object in a multiplexer set.

Parameters

w	Waitable to modify.
event	Descriptor specifying the events that will be polled on the waitable, and the data to be returned when those events are detected.

Returns

0 on success, or a negative error code:
-EINVAL `w` has not been added to a multiplexer set.

See Also

zf_muxer_add().

Note

This function can be used to re-arm waitable after it is returned by zf_muxer_wait() if user likes something like level-triggered events:

```
**   zf_muxer_mod(w, zf_waitable_event(w));
**
```

11.2.2.6 int zf_muxer_wait (struct zf_muxer_set * *muxer*, struct epoll_event * *events*, int *maxevents*, int64_t *timeout_ns*)

Polls a multiplexer set.

Parameters

muxer	Multiplexer set.
events	Array into which to return event descriptors.
maxevents	Maximum number of events to return.
timeout_ns	Maximum time in nanoseconds to block.

Returns

Number of events. Negative values are reserved for future use as error codes, but are not returned at present.

This function polls a multiplexer set and populates an array of event descriptors representing the waitables in that set that are ready. The `events` member of each descriptor specifies the events for which the waitable is actually ready, and the `data` member is set to the user-data associated with that descriptor, as specified in the call to zf_muxer_add() or zf_muxer_mod().

Before checking for ready objects, the function calls zf_reactor_perform() on the set's stack in order to process events from the hardware. In contrast to the rest of the API, zf_muxer_wait() can block. The maximum time to block is specified by `timeout_ns`, and a value of zero results in non-blocking behaviour. A negative value for `timeout_ns` will allow the function to block indefinitely. If the function blocks, it will call zf_reactor_perform() repeatedly in a tight loop.

The multiplexer only supports edge-triggered events: that is, if zf_muxer_wait() reports that a waitable is ready, it need not do so again until a *new* event occurs on tha waitable, even if the waitable is in fact ready. On the other hand, a waitable *may* be reported as ready even when a new event has not occurred, but only when the waitable is in fact ready. A transition from "not ready" to "ready" always constitutes an edge, and in particular, for `EPOLLIN`, the arrival of any new data constitutes an edge.

By default this function has relatively high CPU overhead when no events are ready to be processed and timeout_ns==0, because it polls repeatedly for events. The amount of time spent polling is controlled by stack attribute reactor_spin_count. Setting reactor_spin_count to 1 disables polling and minimises the cost of zf_muxer_wait(timeout_ns=0).

11.2.2.7 struct epoll_event∗ zf_waitable_event (struct zf_waitable ∗ *w*)

Find out the epoll_event data in use with this waitable.

Parameters

w	Waitable to explore.

Returns

The event data.

Note

Function behaviour is undefined if the waitable is not a member of any multiplexer set.

11.2.2.8 int zf_waitable_fd_get (struct zf_stack ∗ *stack,* int ∗ *fd*)

Create an fd that can be used within an epoll set or other standard muxer.

Parameters

stack	Stack the fd should indicate activity for
fd	Updated on success to contain the fd to use

Returns

0 on success, or a negative error code. The possible error-codes are returned from system calls and are system-dependent.

This function creates a file descriptor that can be used within an epoll set (or other standard muxer such as poll or select) to be notified when there is activity on the corresponding stack.

The fd supplied may indicate readiness for a variety of reasons not directly related to the availability of data on a zocket. For example, there is an event that needs processing, a timer has expired, or a connection has changed state. When this occurs the caller should ensure they call zf_muxer_wait() to allow the required activity to take place, and discover if this affected any of the stack's zockets that the caller is interested in. This may or may not result in a zocket within the stack becoming readable or writeable.

Freeing the zf_stack will release all the resources associated with this fd, so it must not be used afterwards. You do not need to call close() on the supplied fd, it will be closed when the stack is freed as part of the zf_stack_free() call.

11.2.2.9 int zf_waitable_fd_prime (struct zf_stack ∗ *stack*)

Prime the fd before blocking.

Parameters

stack	Stack that matches the fd

Returns

0 on success, or a negative error code. The possible error-codes are returned from system calls and are system-dependent.

This primes an fd previously allocated with zf_waitable_fd_get() so it is ready for use with a standard muxer like epoll_wait. The fd should be primed in this way each time the caller blocks waiting for activity.

11.3 types.h File Reference

TCPDirect types.

11.3.1 Detailed Description

TCPDirect types.

Definition in file types.h.

11.4 x86.h File Reference

TCPDirect x86-specific definitions.

11.4.1 Detailed Description

TCPDirect x86-specific definitions. This file contains system-dependent code that is usedthat is used by the other header files. It has no end-user API.

Definition in file x86.h.

11.5 zf.h File Reference

TCPDirect top-level API.

11.5.1 Detailed Description

TCPDirect top-level API. This file should be included in TCPDirect clients. It includes any other TCPDirect header files that are required.

Definition in file zf.h.

11.6 zf_alts.h File Reference

TCPDirect Alternative Sends API.

Typedefs

- typedef uint64_t zf_althandle

    Opaque handle for an alternative.

Functions

- int zf_alternatives_alloc (struct zf_stack ∗stack, const struct zf_attr ∗attr, zf_althandle ∗alt_out)

    Acquire an ID for an alternative queue.
- int zf_alternatives_release (struct zf_stack ∗stack, zf_althandle alt)

    Release an ID for an alternative queue.
- int zf_alternatives_send (struct zf_stack ∗stack, zf_althandle alt)

    Select an alternative and send those messages.
- int zf_alternatives_cancel (struct zf_stack ∗stack, zf_althandle alt)

    Cancel an alternative.
- int zft_alternatives_queue (struct zft ∗ts, zf_althandle alt, const struct iovec ∗iov, int iov_cnt, int flags)

    Queue a TCP message for sending.
- unsigned zf_alternatives_free_space (struct zf_stack ∗stack, zf_althandle alt)

    Query the amount of free buffering on an alt.
- int zf_alternatives_query_overhead_tcp (struct zft ∗ts, struct ef_vi_transmit_alt_overhead ∗out)

    Query TCP per-packet overhead parameters.

11.6.1 Detailed Description

TCPDirect Alternative Sends API.

Definition in file zf_alts.h.

11.6.2 Function Documentation

11.6.2.1 int zf_alternatives_alloc (struct zf_stack * *stack,* const struct zf_attr * *attr,* zf_althandle * *alt_out*)

Acquire an ID for an alternative queue.

Parameters

stack	Stack to allocate the alternative for
attr	Requested attributes for the alternative. At the present time, the attributes are unused. Refer to the attribute documentation in Attributes for details.
alt_out	Handle for the allocated alternative

Returns

0 Success
-ENOMEM No alternative queues available

The alternative queue is identified by opaque handles, and is only able to be used with zockets in the stack provided to this function.

The number of alternatives available to a stack is controlled by the value of the `alt_count` attribute used when creating the stack. This value defaults to zero.

See Also

zf_alternatives_release()

11.6.2.2 int zf_alternatives_cancel (struct zf_stack * *stack,* zf_althandle *alt*)

Cancel an alternative.

Parameters

stack	Stack the alternative was allocated on
alt	Selected alternative

Returns

0 Success

Drops messages queued on this alternative without sending.

You can reuse the alternative queue immediately for new messages (including messages on a different zocket from the previous use) but zft_alternatives_queue() may return -EBUSY until the cancel operation is completed.

11.6.2.3 unsigned zf_alternatives_free_space (struct zf_stack * *stack,* zf_althandle *alt*)

Query the amount of free buffering on an alt.

Parameters

stack	Stack the alternative was allocated on
alt	Selected alternative

Returns

Number of bytes available

The return value of this function is the payload size in bytes of the largest packet which can be sent into this alternative at this moment. Larger packets than this will cause -ENOMEM errors from functions which queue data on alternatives.

Due to per-packet and other overheads, this amount may be different on different alternatives, and is not guaranteed to rise and fall by exactly the sizes of packets queued and sent.

The returned value includes all packet headers. The maximum length of data accepted by zft_alternatives_queue() will be lower than this by the size of the TCP+IP+Ethernet headers. To find a zocket's header size, use zft_get_header_size() or zfut_get_header_size().

11.6.2.4 int zf_alternatives_query_overhead_tcp (struct zft ∗ *ts*, struct ef_vi_transmit_alt_overhead ∗ *out*)

Query TCP per-packet overhead parameters.

Parameters

ts	TCP connection to be queried
out	Returned overhead parameters

Returns

0 on success or -EINVAL if this stack doesn't support alternatives.

This function returns a set of parameters which can be used with ef_vi_transmit_alt_usage() to calculate the amount of buffer space used when sending data via TCP, taking into account the space taken up by headers, VLAN tags, IP options etc.

Use of this function in this way assumes that the transmitted data fits entirely into a single TCP packet.

See the documentation for ef_vi_transmit_alt_usage() for more.

11.6.2.5 int zf_alternatives_release (struct zf_stack ∗ *stack*, zf_althandle *alt*)

Release an ID for an alternative queue.

Parameters

stack	Stack to release the alternative for
alt	zf_alternative to release

Returns

0 Success

Releases allocated alternative queue. If any messages are queued on the specified queue they will be flushed without being sent.

See Also

zf_alternatives_alloc()

11.6.2.6 int zf_alternatives_send (struct zf_stack ∗ *stack*, zf_althandle *alt*)

Select an alternative and send those messages.

Parameters

stack	Stack the alternative was allocated on
alt	Selected alternative

Returns

0 Success
-EBUSY Unable to send due to a transient state (e.g. the alternative queue is being refreshed in response to receiving data).
-EINVAL Unable to send due to inconsistent TCP state (e.g. the zocket is not connected, or has been used via the normal send path after queueing messages on this alternative queue)

On success messages queued on the selected alternative are sent. If other alternative queues have messages queued for the same zocket, their headers will now be out of date and you must call zf_alternatives_cancel() on those queues. You are free to reuse this alternative queue, but until it has finished sending the current set of messages calls to zft_alternatives_queue() will return -EBUSY.

11.6.2.7 int zft_alternatives_queue (struct zft ∗ *ts*, zf_althandle *alt*, const struct iovec ∗ *iov*, int *iov_cnt*, int *flags*)

Queue a TCP message for sending.

Parameters

ts	TCP zocket

alt	ID of the queue to push this message to. Must have been allocated via zf_alternatives_alloc()
iov	TCP payload data to send in this message.
iov_cnt	Number of iovecs to send. Currently must be 1.
flags	Reserved for future use; must be zero.

Returns

0 Success
-EAGAIN Unable to queue due to a transient problem, e.g. the TCP send queue is not empty. These errors may remain present for many milliseconds; the caller should decide whether to retry immediately or to perform other work in the meantime.
-EBUSY Unable to queue due to a transient problem, e.g. the alternative queue is still draining from a previous operation. These errors are expected to clear quickly without outside intervention; the caller can react by calling zf_reactor_perform() and retrying the operation.
-EMSGSIZE Enqueuing the message would exceed the total congestion window.
-ENOMEM Unable to queue due to all packet buffers being allocated already.
-ENOBUFS Unable to queue due to a lack of available buffer space, either in TCP Direct or in the NIC hardware.
-EINVAL Invalid parameters. This includes the case where the alternative already has data queued on another zocket.

This function behaves similarly to zft_send(), but doesn't actually put the data on the wire.

For now it is only possible to send a single buffer of data in each call to zft_alternatives_queue(); this function will return -EINVAL if 'iov_cnt' is not equal to 1. Future releases may change this. Multiple messages can be queued for sending on a single alternative by calling zft_alternatives_queue() for each message.

The current implementation limits all messages enqueued on an alternative to be from the same zocket. This may change in future.

In some cases where an alternative is in the middle of an operation such as a send, cancel, etc. this function may return -EBUSY. In this case the caller should process some events and retry.

11.7 zf_platform.h File Reference

TCPDirect platform API.

11.7.1 Detailed Description

TCPDirect platform API. This file contains platform-dependent code that is used by the other header files. It has no end-user API.

Definition in file zf_platform.h.

11.8 zf_reactor.h File Reference

TCPDirect reactor API for processing stack events.

Functions

- int zf_reactor_perform (struct zf_stack *st)

    Process events on a stack.
- int zf_stack_has_pending_work (const struct zf_stack *st)

    Determine whether a stack has work pending.

11.8.1 Detailed Description

TCPDirect reactor API for processing stack events.

Definition in file zf_reactor.h.

11.8.2 Function Documentation

11.8.2.1 int zf_reactor_perform (struct zf_stack * *st*)

Process events on a stack.

Parameters

st	Stack for which to process events.

This function processes events on a stack and performs the necessary handling. These events include transmit and receive events raised by the hardware, and also software events such as TCP timers. Applications must call this function or zf_muxer_wait() frequently for each stack that is in use. Please see Stack polling in the User Guide for further information.

By default this function has relatively high CPU overhead when no events are ready to be processed, because it polls repeatedly for events. The amount of time spent polling is controlled by stack attribute reactor_spin_count. Setting reactor_spin_count to 1 disables polling and minimises the cost of zf_reactor_perform().

Returns

0 if nothing user-visible occurred as a result.
>0 if something user-visible might have occurred as a result.
Here, "something user-visible occurred" means that the event-processing just performed has had an effect that can be seen by another API call: for example, new data might have arrived on a zocket, in which case that data can be retrieved by one of the receive functions. False positives are possible: a value greater than zero indicates to the application that it should process its zockets, but it does not guarantee that this will yield anything new. Finer-grained advertisement of interesting events can be achieved using the multiplexer.

See Also

zf_muxer_wait()

11.8.2.2 int zf_stack_has_pending_work (const struct zf_stack * *st*)

Determine whether a stack has work pending.

Parameters

st	Stack to check for pending work.

This function returns non-zero if the stack has work pending, and therefore the application should call zf_reactor_perform() or zf_muxer_wait().

This function can be called concurrently with other calls on a stack, and so can be used to avoid taking a serialisation lock (and therefore avoid inducing lock contention) when there isn't any work to do.

Returns

0 if there is nothing to do.
>0 if there is some work pending.

See Also

zf_reactor_perform() zf_muxer_wait()

11.9 zf_stack.h File Reference

TCPDirect stack API.

Macros

- #define EPOLLSTACKHUP EPOLLRDHUP

  Event indicating stack quiescence.

Functions

- int zf_init (void)

  Initialize zf library.
- int zf_deinit (void)

  Deinitialize zf library.
- int zf_stack_alloc (struct zf_attr ∗attr, struct zf_stack ∗∗stack_out)

  Allocate a stack with the supplied attributes.
- int zf_stack_free (struct zf_stack ∗stack)

  Free a stack previously allocated with zf_stack_alloc().
- struct zf_waitable ∗ zf_stack_to_waitable (struct zf_stack ∗)

  Returns a waitable object representing the quiescence of a stack.
- int zf_stack_is_quiescent (struct zf_stack ∗)

  Returns a boolean value indicating whether a stack is quiescent.
- void zf_version (void)

  Print library name and version to stderr.

11.9.1 Detailed Description

TCPDirect stack API.

Definition in file zf_stack.h.

11.9.2 Macro Definition Documentation

11.9.2.1 #define EPOLLSTACKHUP EPOLLRDHUP

Event indicating stack quiescence.

See Also

zf_stack_to_waitable()

Definition at line 85 of file zf_stack.h.

11.9.3 Function Documentation

11.9.3.1 int zf_deinit (void)

Deinitialize zf library.

Returns

0. Negative values are reserved for future use as error returns.

11.9.3.2 int zf_init (void)

Initialize zf library.

Returns

0 on success, or a negative error code. This function uses attributes internally and can return any of the error codes returned by zf_attr_alloc(). Additionally, it can return the following:
-ENOENT Failed to initialize control plane. A likely cause is that Onload drivers are not loaded.

11.9.3.3 int zf_stack_alloc (struct zf_attr ∗ *attr,* struct zf_stack ∗∗ *stack_out*)

Allocate a stack with the supplied attributes.

Parameters

attr	A set of properties to apply to the stack.
stack_out	A pointer to the newly allocated stack.

A stack encapsulates hardware and protocol state. A stack binds to a single network interface, specified by the `interface` attribute in `attr`. To process events on a stack, call zf_reactor_perform() or zf_muxer_wait().

Relevant attributes to set in `attr` are those in the `zf_stack`, `zf_pool` and `zf_vi` categories described in the attributes documentation in Attributes.

Returns

0 on success, or a negative error code:
-EBUSY Out of VI instances or resources for alternatives.
-EINVAL Attribute out of range.
-ENODEV Interface was not specified or was invalid.
-ENOENT Failed to initialize ef_vi or Onload libraries. A likely cause is that Onload drivers are not loaded.
-ENOKEY Adapter is not licensed for TCPDirect.
-ENOMEM Out of memory. N.B. Huge pages are required.
-ENOSPC Out of PIO buffers.
Errors from system calls are also possible. Please consult your system's documentation for `errno(3)`.

11.9.3.4 int zf_stack_free (struct zf_stack ∗ *stack*)

Free a stack previously allocated with zf_stack_alloc().

Parameters

stack	Stack to free

Returns

When called with a valid stack, this function always returns zero. Results on invalid stacks are undefined.

11.9.3.5 int zf_stack_is_quiescent (struct zf_stack ∗)

Returns a boolean value indicating whether a stack is quiescent.

A stack is quiescent precisely when all of the following are true:

- the stack will not transmit any packets except in response to external stimuli (including relevant API calls),
- closing zockets will not result in the transmission of any packets, and
- (optionally, controlled by the `tcp_wait_for_time_wait` stack attribute) there are no TCP zockets in the TIME_WAIT state. In practice, this is equivalent altogether to the condition that there are no open TCP connections.

This can be used to ensure that all connections have been closed gracefully before destroying a stack (or exiting the application). Destroying a stack while it is not quiescent is permitted by the API, but when doing so there is no guarantee that sent data has been acknowledged by the peer or even transmitted, and there is the possibility that peers' connections will be reset.

See Also

zf_stack_to_waitable()

Returns

Non-zero if the stack is quiescent, or zero otherwise.

11.9.3.6 struct zf_waitable∗ zf_stack_to_waitable (struct zf_stack ∗)

Returns a waitable object representing the quiescence of a stack.

The waitable will be ready for EPOLLSTACKHUP if the stack is quiescent.

See Also

zf_stack_is_quiescent()

Returns

Waitable.

11.10 zf_tcp.h File Reference

TCPDirect TCP API.

```
#include <netinet/in.h>
#include <sys/uio.h>
```

Data Structures

- struct zftl

    Opaque structure describing a TCP listening zocket.
- struct zft

    Opaque structure describing a TCP zocket that is connected.
- struct zft_msg

    TCP zero-copy RX message structure.

Functions

- int zftl_listen (struct zf_stack ∗st, const struct sockaddr ∗laddr, socklen_t laddrlen, const struct zf_attr ∗attr, struct zftl ∗∗tl_out)

    Allocate TCP listening zocket.
- int zftl_accept (struct zftl ∗tl, struct zft ∗∗ts_out)

    Accept incoming TCP connection.
- struct zf_waitable ∗ zftl_to_waitable (struct zftl ∗tl)

    Returns a `zf_waitable` representing `tl`.
- void zftl_getname (struct zftl ∗ts, struct sockaddr ∗laddr_out, socklen_t ∗laddrlen)

    Retrieve the local address of the zocket.
- int zftl_free (struct zftl ∗ts)

    Release resources associated with a TCP listening zocket.
- struct zf_waitable ∗ zft_to_waitable (struct zft ∗ts)

    Returns a zf_waitable representing the given zft.
- int zft_alloc (struct zf_stack ∗st, const struct zf_attr ∗attr, struct zft_handle ∗∗handle_out)

Allocate active-open TCP zocket.

- int zft_handle_free (struct zft_handle *handle)

  Release a handle to a TCP zocket.
- void zft_handle_getname (struct zft_handle *ts, struct sockaddr *laddr_out, socklen_t *laddrlen)

  Retrieve the local address to which a zft_handle is bound.
- int zft_addr_bind (struct zft_handle *handle, const struct sockaddr *laddr, socklen_t laddrlen, int flags)

  Bind to a specific local address.
- int zft_connect (struct zft_handle *handle, const struct sockaddr *raddr, socklen_t raddrlen, struct zft **ts_-out)

  Connect a TCP zocket.
- int zft_shutdown_tx (struct zft *ts)

  Shut down outgoing TCP connection.
- int zft_free (struct zft *ts)

  Release resources associated with a TCP zocket.
- int zft_state (struct zft *ts)

  Return the TCP state of a TCP zocket.
- int zft_error (struct zft *ts)

  Find out the error type happened on the TCP zocket.
- void zft_getname (struct zft *ts, struct sockaddr *laddr_out, socklen_t *laddrlen, struct sockaddr *raddr_out, socklen_t *raddrlen)

  Retrieve the local address of the zocket.
- void zft_zc_recv (struct zft *ts, struct zft_msg *msg, int flags)

  Zero-copy read of available packets.
- int zft_zc_recv_done (struct zft *ts, struct zft_msg *msg)

  Concludes pending zc_recv operation as done.
- int zft_zc_recv_done_some (struct zft *ts, struct zft_msg *msg, size_t len)

  Concludes pending zc_recv operation as done acknowledging all or some of the data to have been read.
- int zft_recv (struct zft *ts, const struct iovec *iov, int iovcnt, int flags)

  Copy-based receive.
- ssize_t zft_send (struct zft *ts, const struct iovec *iov, int iov_cnt, int flags)

  Send data specified in iovec array.
- ssize_t zft_send_single (struct zft *ts, const void *buf, size_t buflen, int flags)

  Send data given in single buffer.
- int zft_send_space (struct zft *ts, size_t *space)

  Query available space in the send queue.
- int zft_get_mss (struct zft *ts)

  Retrieve the maximum segment size (MSS) for a TCP connection.
- unsigned zft_get_header_size (struct zft *ts)

  Return protocol header size for this connection.

11.10.1 Detailed Description

TCPDirect TCP API.

Definition in file zf_tcp.h.

11.10.2 Function Documentation

11.10.2.1 int zft_addr_bind (struct zft_handle * *handle,* const struct sockaddr * *laddr,* socklen_t *laddrlen,* int *flags*)

Bind to a specific local address.

Parameters

handle	TCP zocket handle.
laddr	Local address.
laddrlen	Length of structure pointed to by laddr
flags	Reserved. Must be zero.

Returns

0 Success.
-EADDRINUSE Local address already in use.
-EADDRNOTAVAIL `laddr` is not a local address.
-EAFNOSUPPORT `laddr` is not an AF_INET address.
-EFAULT Invalid pointer.
-EINVAL Zocket is already bound, invalid `flags`, or invalid laddrlen.
-ENOMEM Out of memory.

11.10.2.2 int zft_alloc (struct zf_stack * *st*, const struct zf_attr * *attr*, struct zft_handle ** *handle_out*)

Allocate active-open TCP zocket.

Parameters

st	Initialized zf_stack.
attr	Attributes required for this TCP zocket. Note that not all attributes are relevant; only those which apply to objects of type "zf_socket" are applicable here. Refer to the Attributes documentation for details.
handle_out	On successful return filled with pointer to a zocket handle. This handle can be used to refer to the zocket before it is connected.

Returns

0 Success.
-ENOBUFS No zockets of this type available.

This function initialises the datastructures needed to make an outgoing TCP connection.

The returned handle can be used to refer to the zocket before it is connected.

The handle must be released either by explicit release with zft_handle_free(), or by conversion to a connected zocket via zft_connect().

See Also

zft_addr_bind() zft_connect() zft_handle_free()

11.10.2.3 int zft_connect (struct zft_handle * *handle*, const struct sockaddr * *raddr*, socklen_t *raddrlen*, struct zft ** *ts_out*)

Connect a TCP zocket.

Parameters

handle	TCP zocket handle, to be replaced by the returned zocket.
raddr	Remote address to connect to.
raddrlen	Length of structure pointed to by raddr.
ts_out	On successful return, a pointer to a TCP zocket.

This replaces the zocket handle with a TCP zocket. On successful return the zocket handle has been released and is no longer valid.

If a specific local address has not been set via zft_addr_bind() then an appropriate one will be selected.

This function does not block. Functions that attempt to transfer data on the zocket between zft_connect() and the successful establishment of the underlying TCP connection will return an error. Furthermore, failure of the remote host to accept the connection will not be reported by this function, but instead by any attempts to read from the zocket (or by zft_error()). As such, after calling zft_connect(), either

- read calls that fail with `-ENOTCONN` should be repeated after calling zf_reactor_perform(), or
- the zocket should be polled for readiness using zf_muxer_wait().

This is analogous to the non-blocking connection model for POSIX sockets.

Returns

0 Success.
-EAFNOSUPPORT raddr is not an AF_INET address
-EADDRINUSE Address already in use.
-EBUSY Out of hardware resources.
-EFAULT Invalid pointer.
-EHOSTUNREACH No route to remote host.
-ENOMEM Out of memory.

See Also

zft_addr_bind()

11.10.2.4 int zft_error (struct zft * *ts*)

Find out the error type happened on the TCP zocket.

Parameters

ts	TCP zocket.

Return values

errno	value, similar to SO_ERROR value for sockets.

Error	values are designed to be similar to Linux SO_ERROR:
ECONNREFUSED	The connection attempt was refused by server.
ECONNRESET	The connection was reset by the peer after it was established.
ETIMEDOUT	The connection was timed out, probably because of network failure.
EPIPE	The connection was closed gracefully by the peer (i.e. we've received all the data they've sent to us), but the peer refused to receive the data we've tried to send.

11.10.2.5 int zft_free (struct zft ∗ *ts*)

Release resources associated with a TCP zocket.

Parameters

ts	TCP zocket.

This call shuts down the zocket if necessary. The application must not use `ts` after this call.

Returns

0 on success. Negative values are reserved for future use as error codes, but are not returned at present.

11.10.2.6 unsigned zft_get_header_size (struct zft ∗ *ts*)

Return protocol header size for this connection.

Parameters

ts	The TCP zocket to query the header size for.

Returns

Protocol header size in bytes.

This function returns the total size of all protocol headers in bytes. An outgoing packet's size will be exactly the sum of this value and the number of payload data bytes it contains.

This function cannot fail.

11.10.2.7 int zft_get_mss (struct zft ∗ *ts*)

Retrieve the maximum segment size (MSS) for a TCP connection.

Parameters

ts	The TCP zocket to query.

Returns

>= 0 The value of the MSS in bytes.
-ENOTCONN Zocket is not in a valid TCP state for sending.

11.10.2.8 void zft_getname (struct zft ∗ *ts,* struct sockaddr ∗ *laddr_out,* socklen_t ∗ *laddrlen,* struct sockaddr ∗ *raddr_out,* socklen_t ∗ *raddrlen*)

Retrieve the local address of the zocket.

Parameters

ts	TCP zocket.
laddr_out	Return the local address of the zocket.
laddrlen	The length of the structure pointed to by laddr_out
raddr_out	Return the remote address of the zocket.
raddrlen	The length of the structure pointed to by raddr_out

This function returns local and/or remote IP address and TCP port of the given connection. Caller may pass NULL pointer for local or remote address if he is interested in the other address only.

If the supplied address structures are too small the result will be truncated and addrlen updated to a length greater than that supplied.

11.10.2.9 int zft_handle_free (struct zft_handle * *handle*)

Release a handle to a TCP zocket.

Parameters

handle	Handle to be released.

This function releases resources associated with a zft_handle.

Returns

0 Success.

11.10.2.10 void zft_handle_getname (struct zft_handle * *ts*, struct sockaddr * *laddr_out*, socklen_t * *laddrlen*)

Retrieve the local address to which a zft_handle is bound.

Parameters

ts	TCP zocket handle
laddr_out	Return the local address of the zocket
laddrlen	On entry, the size in bytes of the structure pointed to by laddr_out. Set on return to be the size in bytes of the result.

This function returns the local IP address and TCP port of the given listener. The behavior is undefined if the zocket is not bound.

If the supplied structure is too small the result will be truncated and laddrlen updated to a length greater than that supplied.

11.10.2.11 int zft_recv (struct zft * *ts*, const struct iovec * *iov*, int *iovcnt*, int *flags*)

Copy-based receive.

Parameters

ts	TCP zocket
iov	Array with vectors pointing to buffers to fill with packet payloads.
iovcnt	The maximum number of buffers supplied (i.e. size of `iov`)
flags	None yet, must be zero.

Returns

$>$0 Number of bytes successfully received
0 End of File - other end has closed the connection
-EAGAIN No data avaible to read.
Other error codes are as for zft_zc_recv_done().

Copies received data on a zocket into buffers provided by the caller. The number of bytes received is returned. The caller's buffers will be filled as far as possible, and so a positive return value of less than the total space available in `iov` implies that no further data is available.

If no data is available, there are two possibilities: either the connection is still open, in which case `-EAGAIN` is returned, or else the connection has been closed by the peer, in which case the function succeeds and returns zero.

11.10.2.12 ssize_t zft_send (struct zft * *ts*, const struct iovec * *iov*, int *iov_cnt*, int *flags*)

Send data specified in iovec array.

Parameters

ts	The TCP zocket to send on.
iov	The iovec of data to send.
iov_cnt	The length of iov.
flags	Flags. 0 or MSG_MORE.

Sends the supplied data or its part. MSG_MORE flag will prevent sending packet that is not filled up to MSS.

Provided buffers may be re-used on successful return from this function.

Returns

Number of bytes sent on success.
-EINVAL Incorrect arguments supplied.
-ENOTCONN Zocket is not in a valid TCP state for sending.
-EAGAIN Not enough space (either bytes or buffers) in the send queue.
-ENOMEM Not enough packet buffers available.

Note

This function does not support sending zero-length data, and does not raise an error if you do so. Every iovec in the iov array must have length greater than 0, and iov_cnt must also be greater than 0.
The flags argument must be set to 0 or MSG_MORE.
Notes on current implementation:

1. Currently, this function will return `-ENOMEM` without sending any data if it is unable to send the entire message due to shortage of packet buffers. This behaviour might change in future releases.

2. In case of partial send, the data is queued with MSG_MORE flag set, and so may not go out immediately. See below for details of how to flush a MSG_MORE send.
3. MSG_MORE flag prevents the last partially filled segment from being sent immediately. The only guaranteed way to flush such a segment is to follow MSG_MORE send with normal send - otherwise the segment might never get sent at all or it may take undefined amount of time. Some non-guaranteed triggers that might induce flush of a MSG_MORE segment:
   - further MSG_MORE send causes the segment to become full,
   - preceding normal send left paritally filled segment in sendqueue, or
   - during stack polling TCP state machine intends to send ACK in response to incoming data.

11.10.2.13 ssize_t zft_send_single (struct zft ∗ *ts,* const void ∗ *buf,* size_t *buflen,* int *flags*)

Send data given in single buffer.

Parameters

ts	The TCP zocket to send on.
buf	The buffer of data to send.
buflen	The length of buffer.
flags	Flags. 0 or MSG_MORE.

Sends the supplied data or its part. MSG_MORE flag will prevent sending packet that is not filled up to MSS.

Provided buffer may be re-used on successful return from this function.

Returns

Number of bytes sent on success.
-EINVAL Incorrect arguments supplied.
-ENOTCONN Zocket is not in a valid TCP state for sending.
-EAGAIN Not enough space (either bytes or buffers) in the send queue.
-ENOMEM Not enough packet buffers available.

Note

This function does not support sending zero-length data, and does not raise an error if you do so.
The flags argument must be set to 0 or MSG_MORE.
Notes on current implementation:

1. Currently, this function will return `-ENOMEM` without sending any data if it is unable to send the entire message due to shortage of packet buffers. This behaviour might change in future releases.
2. MSG_MORE flag prevents the last partially filled segment from being sent immediately. The only guaranteed way to flush such a segment is to follow MSG_MORE send with normal send - otherwise the segment might never get sent at all or it may take undefined amount of time. Some non-guaranteed triggers that might induce flush of a MSG_MORE segment:
   - further MSG_MORE send causes the segment to become full,
   - preceding normal send left paritally filled segment in sendqueue, or
   - during stack polling TCP state machine intends to send ACK in response to incoming data.

11.10.2.14 int zft_send_space (struct zft ∗ *ts,* size_t ∗ *space*)

Query available space in the send queue.

Parameters

ts	The TCP zocket to query the send queue for.
space	On successful return, the available space in bytes.

This function will return the current space available in the send queue for the given zocket. This can be used to avoid zft_send() returning `-EAGAIN`.

Returns

0 Success.
-ENOTCONN Zocket is not in a valid TCP state for sending.

Note

Available send queue space is a function of the number of the number of bytes queued, the number of internal buffers in the queue, and the MSS. Making many small sends can therefore consume more space than a single large send, and force zft_send() to compress the send queue to avoid returning `-EAGAIN`.

11.10.2.15 int zft_shutdown_tx (struct zft * *ts*)

Shut down outgoing TCP connection.

Parameters

ts	A connected TCP zocket.

This function closes the TCP connection, preventing further data transmission except for already-queued data. This function does not prevent the connection from receiving more data.

Returns

0 on success, or a negative error code. Error codes returned are similar to zft_send() ones:
-ENOTCONN Inappropriate TCP state: not connected or already shut down.
-EAGAIN Not enough space (either bytes or buffers) in the send queue.
-ENOMEM Not enough packet buffers available.

11.10.2.16 int zft_state (struct zft * *ts*)

Return the TCP state of a TCP zocket.

Parameters

ts	TCP zocket.

Returns

Standard `TCP_*` state constant (e.g. `TCP_ESTABLISHED`).

11.10.2.17 struct zf_waitable* zft_to_waitable (struct zft * *ts*)

Returns a zf_waitable representing the given zft.

Parameters

ts	The zft to return as a zf_waitable

Returns

The zf_waitable

This function is necessary to use TCP zockets with the multiplexer.

11.10.2.18 void zft_zc_recv (struct zft ∗ *ts,* struct zft_msg ∗ *msg,* int *flags*)

Zero-copy read of available packets.

Parameters

ts	TCP zocket.
msg	Message structure.
flags	Reserved. Must be zero.

This function completes the supplied `msg` structure with details of received packet buffers. In case of EOF a zero-length buffer is appended at the end of data stream and to identify reason of stream termination check result of zft_zc_recv_done() or of zft_zc_recv_done_some().

The function will only fill fewer iovecs in `msg` than are provided in the case where no further data is available.

Buffers are 'locked' until zft_zc_recv_done() or zft_zc_recv_done_some() is performed. The caller must not modify the contents of `msg` until after it has been passed to zft_zc_recv_done() or to zft_zc_recv_done_some().

11.10.2.19 int zft_zc_recv_done (struct zft ∗ *ts,* struct zft_msg ∗ *msg*)

Concludes pending zc_recv operation as done.

Parameters

ts	TCP zocket
msg	Message

Returns

$>=$ 1 Connection still receiving.
0 EOF.
-ECONNREFUSED Connection refused. This is possible as zft_connect() is non-blocking.
-ECONNRESET Connection reset by peer.
-EPIPE Peer closed connection gracefully, but refused to receive some data sent on this zocket.
-ETIMEDOUT Connection timed out.

This function (or zft_zc_recv_done_some()) must be called after each successful zft_zc_recv() operation that returned at least one packet. It must not be called otherwise (in particular, when zft_zc_recv() returned no packets). The function releases resources and enables subseqent calls to zft_zc_recv() or zft_recv(). `msg` must be passed unmodified from the call to zft_zc_recv().

11.10.2.20 int zft_zc_recv_done_some (struct zft ∗ *ts,* struct zft_msg ∗ *msg,* size_t *len*)

Concludes pending zc_recv operation as done acknowledging all or some of the data to have been read.

Parameters

ts	TCP zocket.
msg	Message.
len	Total number of bytes read by the client.

Returns

As for zft_zc_recv_done().

Can be called after each successful zft_zc_recv() operation as an alternative to zft_zc_recv_done() or in cases where not all payload have been consumed. The restictions on when it may be called are the same as for zft_zc-_recv_done(). The function releases resources and enables subseqent calls to zft_zc_recv() or zft_recv(). zft_zc-_recv() or zft_recv() functions will return data indicated as non-read when they are called next time. `msg` must be passed unmodified from the call to zft_zc_recv(). `len` must not be greater than total payload returned by zft_zc_-recv().

11.10.2.21 int zftl_accept (struct zftl ∗ *tl*, struct zft ∗∗ *ts_out*)

Accept incoming TCP connection.

Parameters

tl	The listening zocket from which to accept the connection.
ts_out	On successful return filled with pointer to a TCP zocket for the new connection.

Returns

0 Success.
-EAGAIN No incoming connections available.

11.10.2.22 int zftl_free (struct zftl ∗ *ts*)

Release resources associated with a TCP listening zocket.

Parameters

ts	A TCP listening zocket.

This call shuts down the listening zocket, closing any connections waiting on the zocket that have not yet been accepted. The application must not use `ts` after this call.

Returns

0 Success.

11.10.2.23 void zftl_getname (struct zftl ∗ *ts*, struct sockaddr ∗ *laddr_out*, socklen_t ∗ *laddrlen*)

Retrieve the local address of the zocket.

Parameters

ts	TCP zocket.
laddr_out	Set on return to the local address of the zocket.
laddrlen	On entry, the size in bytes of the structure pointed to by laddr_out. Set on return to be the size in bytes of the result.

This function returns the local IP address and TCP port of the listening zocket. If the supplied structure is too small the result will be truncated and laddrlen updated to a length greater than that supplied.

11.10.2.24 int zftl_listen (struct zf_stack * *st*, const struct sockaddr * *laddr*, socklen_t *laddrlen*, const struct zf_attr * *attr*, struct zftl ** *tl_out*)

Allocate TCP listening zocket.

Parameters

st	Initialized `zf_stack` in which to created the listener.
laddr	Local address on which to listen. Must be non-null, and must be a single local address (not INADDR_ANY).
laddrlen	The size in bytes of the structure pointed to by laddr
attr	Attributes to apply to this zocket. Note that not all attributes are relevant; only those which apply to objects of type "zf_socket" are applicable here. Refer to the attribute documentation in Attributes for details.
tl_out	On successful return filled with pointer to created TCP listening zocket.

Returns

0 Success.
-EFAULT Invalid laddr pointer.
-EADDRINUSE Local address already in use.
-EADDRNOTAVAIL `laddr` is not a local address.
-EAFNOSUPPORT `laddr` is not an AF_INET address.
-EINVAL Zocket is already listening, or invalid addr length.
-ENOBUFS No zockets of this type available.
-ENOMEM Out of memory.
-EOPNOTSUPP `laddr` is INADDR_ANY.

11.10.2.25 struct zf_waitable* zftl_to_waitable (struct zftl * *tl*)

Returns a `zf_waitable` representing `tl`.

Parameters

tl	The zftl to return as a zf_waitable

Returns

The zf_waitable

This function is necessary to use TCP listening zockets with the multiplexer.

11.11 zf_udp.h File Reference

TCPDirect UDP API.

```
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <assert.h>
```

Data Structures

- struct zfur

  Opaque structure describing a UDP-receive zocket.
- struct zfur_msg

  UDP zero-copy RX message structure.
- struct zfut

  Opaque structure describing a UDP-transmit zocket.

Macros

- #define ZFUT_FLAG_DONT_FRAGMENT IP_DF /∗ 0x2000∗/

  Flags for zfut_send()

Functions

- int zfur_alloc (struct zfur ∗∗us_out, struct zf_stack ∗st, const struct zf_attr ∗attr)

  Creates UDP-receive zocket.
- int zfur_free (struct zfur ∗us)

  Release UDP-receive zocket previously created with zfur_alloc().
- int zfur_addr_bind (struct zfur ∗us, struct sockaddr ∗laddr, socklen_t laddrlen, const struct sockaddr ∗raddr, socklen_t raddrlen, int flags)

  Configures UDP-receive zocket to receive on a specified address.
- int zfur_addr_unbind (struct zfur ∗us, const struct sockaddr ∗laddr, socklen_t laddrlen, const struct sockaddr ∗raddr, socklen_t raddrlen, int flags)

  Unbind UDP-receive zocket from address.
- void zfur_zc_recv (struct zfur ∗us, struct zfur_msg ∗msg, int flags)

  Zero-copy read of single datagram.
- void zfur_zc_recv_done (struct zfur ∗us, struct zfur_msg ∗msg)

  Concludes pending zero-copy receive operation as done.
- int zfur_pkt_get_header (struct zfur ∗us, const struct zfur_msg ∗msg, const struct iphdr ∗∗iphdr, const struct udphdr ∗∗udphdr, int pktind)

  Retrieves remote address from the header of a received packet.
- struct zf_waitable ∗ zfur_to_waitable (struct zfur ∗us)

  Returns a zf_waitable representing the given zfur.
- int zfut_alloc (struct zfut ∗∗us_out, struct zf_stack ∗st, const struct sockaddr ∗laddr, socklen_t laddrlen, const struct sockaddr ∗raddr, socklen_t raddrlen, int flags, const struct zf_attr ∗attr)

Allocate a UDP-transmit zocket.
- int zfut_free (struct zfut ∗us)

  Free UDP-transmit zocket.
- int zfut_get_mss (struct zfut ∗us)

  Get the maximum segment size which can be transmitted.
- int zfut_send_single (struct zfut ∗us, const void ∗buf, size_t buflen)

  Copy-based send of single non-fragmented UDP packet.
- int zfut_send (struct zfut ∗us, const struct iovec ∗iov, int iov_cnt, int flags)

  Copy-based send of single UDP packet (possibly fragmented).
- struct zf_waitable ∗ zfut_to_waitable (struct zfut ∗us)

  Returns a zf_waitable representing the given zfut.
- unsigned zfut_get_header_size (struct zfut ∗us)

  Return protocol header size for this zocket.

11.11.1 Detailed Description

TCPDirect UDP API.

Definition in file zf_udp.h.

11.11.2 Function Documentation

11.11.2.1 int zfur_addr_bind (struct zfur ∗ *us,* struct sockaddr ∗ *laddr,* socklen_t *laddrlen,* const struct sockaddr ∗ *raddr,* socklen_t *raddrlen,* int *flags*)

Configures UDP-receive zocket to receive on a specified address.

Parameters

us	The zocket to bind
laddr	Local address. Cannot be NULL or INADDR_ANY, but the port may be zero, in which case an ephemeral port is allocated.
laddrlen	Length of the structure pointed to by laddr.
raddr	Remote address. If NULL, traffic will be accepted from all remote addresses.
raddrlen	Length of the structure pointed to by raddr.
flags	Flags. Must be zero.

Returns

0 Success.
-EADDRINUSE Address already in use.
-EAFNOSUPPORT laddr and/or raddr are not AF_INET addresses
-EBUSY Out of hardware resources.
-EINVAL Invalid address length supplied.
-EFAULT Invalid address supplied.
-ENOMEM Out of memory.

The port number in `laddr` is updated if it was set to 0 by the caller.

If the specified local address is multicast then this has the effect of joining the multicast group as well as setting the filter. The group membership will persist until either the address is unbound (see zfur_addr_unbind()), or the zocket is closed.

11.11.2.2 int zfur_addr_unbind (struct zfur ∗ *us,* const struct sockaddr ∗ *laddr,* socklen_t *laddrlen,* const struct sockaddr ∗ *raddr,* socklen_t *raddrlen,* int *flags*)

Unbind UDP-receive zocket from address.

Parameters

us	The zocket to unbind.
laddr	Local address. Can be NULL to match any local address.
laddrlen	Length of the structure pointed to by laddr.
raddr	Remote address. Can be NULL to match any remote address.
raddrlen	Length of the structure pointed to by raddr.
flags	Flags. Must be zero.

Returns

0 Success.
-EINVAL The zocket is not bound to the specified address.

The addresses specified must match those used in zfur_addr_bind().

11.11.2.3 int zfur_alloc (struct zfur ** *us_out*, struct zf_stack * *st*, const struct zf_attr * *attr*)

Creates UDP-receive zocket.

Parameters

us_out	Pointer to receive new UDP-receive zocket's address.
st	Initialized zf_stack in which to create the zocket.
attr	Attributes to apply to this zocket. Note that not all attributes are relevant; only those which apply to objects of type "zf_socket" are applicable here. Refer to the attribute documentation in Attributes for details.

Returns

0 Success.
-ENOBUFS No zockets of this type available.

Associates UDP-receive zocket with semi-wild or full hardware filter. Creates software filter and initializes receive queue. The zocket becomes ready to receive packets after this call.

11.11.2.4 int zfur_free (struct zfur * *us*)

Release UDP-receive zocket previously created with zfur_alloc().

Parameters

us	The UDP zocket to release.

Returns

0 on success. Negative values are reserved for future use as error codes, but are not returned at present.

11.11.2.5 int zfur_pkt_get_header (struct zfur * *us*, const struct zfur_msg * *msg*, const struct iphdr ** *iphdr*, const struct udphdr ** *udphdr*, int *pktind*)

Retrieves remote address from the header of a received packet.

Parameters

us	UDP zocket.
msg	Message.
iphdr	Location to receive IP header.
udphdr	Location to receive UDP header.
pktind	Index of packet within `msg->iov`.

This is useful for zockets that can receive from many remote addresses, i.e. those for which zfur_addr_bind() was called with `raddr == NULL`.

11.11.2.6 struct zf_waitable* zfur_to_waitable (struct zfur * *us*)

Returns a zf_waitable representing the given zfur.

Parameters

us	The zfur to return as a zf_waitable

Returns

The zf_waitable

This is necessary for use with the multiplexer.

11.11.2.7 void zfur_zc_recv (struct zfur * *us,* struct zfur_msg * *msg,* int *flags*)

Zero-copy read of single datagram.

Parameters

us	UDP zocket.
msg	Message structure.
flags	Must be zero.

This function completes the supplied `msg` structure with details of a received UDP datagram.

The function may not fill all the supplied iovecs in `msg` even in the case where further data is available, but you can discover if there is more data available using the dgrams_left field in zfur_msg after making this call.

TCPDirect does not yet support fragmented datagrams, but in the future such datagrams will be represented in the `msg` iovec as a scatter-gather array of packet buffers. If the iovec is not long enough it may return a partial datagram.

Buffers are 'locked' until zfur_zc_recv_done() is performed. The caller must not modify the contents of `msg` until after it has been passed to zfur_zc_recv_done().

11.11.2.8 void zfur_zc_recv_done (struct zfur * *us,* struct zfur_msg * *msg*)

Concludes pending zero-copy receive operation as done.

Parameters

us	UDP zocket.
msg	Message.

Must be called after each successful zfur_zc_recv() operation that returns at least one packet. It must not be called otherwise (in particular, when zfur_zc_recv() returned no packets). The function releases resources and enables subseqent calls to zfur_zc_recv(). `msg` must be passed unmodified from the call to zfur_zc_recv().

11.11.2.9 int zfut_alloc (struct zfut ∗∗ *us_out*, struct zf_stack ∗ *st*, const struct sockaddr ∗ *laddr*, socklen_t *laddrlen*, const struct sockaddr ∗ *raddr*, socklen_t *raddrlen*, int *flags*, const struct zf_attr ∗ *attr*)

Allocate a UDP-transmit zocket.

Parameters

us_out	On success contains pointer to newly created UDP transmit zocket
st	Stack in which to create zocket
laddr	Local address. If INADDR_ANY is specified, the local address will be selected according to the route to `raddr`, but the port must be non-zero.
laddrlen	Length of the structure pointed to by laddr.
raddr	Remote address.
raddrlen	Length of the structure pointed to by raddr.
flags	Must be zero.
attr	Attributes to apply to the zocket. Note that not all attributes are relevant; only those which apply to objects of type "zf_socket" are applicable here. Refer to the attribute documentation in Attributes for details.

Returns

0 Success.
-EFAULT Invalid pointer.
-EHOSTUNREACH No route to remote host.
-EINVAL Invalid local or remote address, or address lengths.
-ENOBUFS No zockets of this type available.

Note

Once the zocket is created, neither the local address nor the remote address can be changed.

11.11.2.10 int zfut_free (struct zfut ∗ *us*)

Free UDP-transmit zocket.

Parameters

us	UDP-transmit zocket to free.

Returns

0 on success. Negative values are reserved for future use as error codes, but are not returned at present.

11.11.2.11 unsigned zfut_get_header_size (struct zfut ∗ *us*)

Return protocol header size for this zocket.

Parameters

us	The UDP-TX zocket to query the header size for.

Returns

Protocol header size in bytes.

This function returns the total size of all protocol headers in bytes. An outgoing packet's size will be exactly the sum of this value and the number of payload data bytes it contains.

This function cannot fail.

11.11.2.12 int zfut_get_mss (struct zfut * *us*)

Get the maximum segment size which can be transmitted.

Returns

Maximum buflen parameter which can be passed to zfut_send_single(). This value is constant for a given zocket.

11.11.2.13 int zfut_send (struct zfut * *us*, const struct iovec * *iov*, int *iov_cnt*, int *flags*)

Copy-based send of single UDP packet (possibly fragmented).

Parameters

us	The UDP zocket to send on.
iov	The iovec of data to send.
iov_cnt	The length of iov.
flags	Flags.

Returns

Payload bytes sent (i.e. `buflen`) on success.
-EAGAIN Hardware queue full. Call zf_reactor_perform() until it returns non-zero and try again.
-EMSGSIZE Message too large.
-ENOBUFS Out of packet buffers.

For a small packet in a plain buffer with the ZFUT_FLAG_DONT_FRAGMENT flag set, this function just calls zfut_send_single(). Otherwise it handles IO vector and fragments a UDP packet into multiple IP fragments as needed.

If ZFUT_FLAG_DONT_FRAGMENT flag is specified, then the datagram should fit to the MSS value (see zfut_get_mss() above), and the DontFragment bit in the IP header will be set.

See Also

zfut_send_single()

11.11.2.14 int zfut_send_single (struct zfut * *us*, const void * *buf*, size_t *buflen*)

Copy-based send of single non-fragmented UDP packet.

Parameters

us	The UDP zocket to send on.
buf	A buffer of the data to send.
buflen	The length of the buffer, in bytes.

Returns

Payload bytes sent (i.e. `buflen`) on success.
-EAGAIN Hardware queue full. Call zf_reactor_perform() until it returns non-zero and try again.
-ENOBUFS Out of packet buffers.

The function uses PIO when possible (i.e. for small datagrams), and always sets the DontFragment bit in the IP header. `buflen` must be no larger than the value returned by zfut_get_mss().

See Also

zfut_get_mss() zfut_send()

11.11.2.15 struct zf_waitable* zfut_to_waitable (struct zfut * *us*)

Returns a zf_waitable representing the given zfut.

Parameters

us	The zfut to return as a zf_waitable.

Returns

The zf_waitable.

This function is necessary to use UDP-transmit zockets with the multiplexer.

Index