ZeroTier SDK

Connect physical devices, virtual devices, and application instances as if everything is on a single LAN.

The ZeroTier SDK brings your network into userspace. We've paired our network hypervisor core with a network stack (lwIP) to provide your application with an exclusive and private virtual network interface. All traffic on this interface is end-to-end encrypted between each peer and we provide an easy-to-use socket interface derived from Berkeley Sockets. Since we aren't using the kernel's network stack that means, no drivers, no root, and no host configuration requirements. For a more in-depth discussion on the technical side of ZeroTier, check out our Manual. For troubleshooting advice see our Knowledgebase. If you need further assistance, create an account at my.zerotier.com and join our community of users and professionals.

Usage

Downloads and examples: download.zerotier.com/dist/sdk

Homebrew

```
brew install libzt
clang++ -o yourApp yourApp.cpp -lzt; ./yourApp
```

Building from source

To build both `release` and `debug` libraries for only your host's architecture use `make host`. Or optionally `make host_release` for release only. To build everything including things like iOS frameworks, Android packages, etc, use `make all`. Possible build targets can be seen by using `make list`. Resultant libraries will be placed in `./lib`, test and example programs will be placed in `./bin`:

```
make update; make patch; make host
```

Typical build output:

```
lib
├── release
|    └── linux-x86_64
|        ├── libzt.a
|        └── libzt.so
└── debug
    └── linux-x86_64
        ├── libzt.a
        └── libzt.so
bin
└── release
    └── linux-x86_64
        ├── adhoc
        ├── client
        ├── comprehensive
        └── server
```

Example linking step:

```
clang++ -o yourApp yourApp.cpp -L./lib/release/linux-x86_64/ -lzt; ./yourApp
```

Starting the service

The next few sections explain how to use the network control interface portion of the API. These functions are non-blocking and will return an error code specified in the **Error handling** section and will result in the generation of callback events detailed in the **Event handling** section. It is your responsibility to handle these events. To start the service, simply call:

`zts_start(char *path, void (*userCallbackFunc)(struct zts_callback_msg*), int port)`

At this stage, if a cryptographic identity for this node does not already exist on your local storage medium, it will generate a new one and store it, the node's address (commonly referred to as `nodeId`) will be derived from this identity and will be presented to you upon receiving the `ZTS_EVENT_NODE_ONLINE` shown below. The first argument `path` is a path where you will direct ZeroTier to store its automatically-generated cryptographic identity files (`identity.public` and `identity.secret`), these files are your keys to communicating on the network. Keep them safe and keep them unique. If any two nodes are online using the same identities you will have a bad time. The second argument `userCallbackFunc` is a function that you specify to handle all generated events for the life of your program, see below:

```
#include "ZeroTier.h"

...

bool networkReady = false;

void myZeroTierEventCallback(struct zts_callback_msg *msg)
{
    if (msg->eventCode == ZTS_EVENT_NODE_ONLINE) {
        printf("ZTS_EVENT_NODE_ONLINE, nodeId=%llx\n", msg->node->address);
        networkReady = true;
    }
    ...
}

int main()
{
    zts_start("configPath", &myZeroTierEventCallback, 9994);
    uint64_t nwid = 0x0123456789abcdef;
    while (!networkReady) { sleep(1); }
    zts_join(nwid);
    int fd = zts_socket(ZTS_AF_INET, ZTS_SOCK_STREAM, 0);
    ...
    return 0;
}
```

For more complete examples see `./examples/`

After calling `zts_start()` you will receive one or more of the following events:

```
ZTS_EVENT_NODE_OFFLINE
ZTS_EVENT_NODE_ONLINE
ZTS_EVENT_NODE_DOWN
ZTS_EVENT_NODE_IDENTITY_COLLISION
ZTS_EVENT_NODE_UNRECOVERABLE_ERROR
ZTS_EVENT_NODE_NORMAL_TERMINATION
```

After receiving `ZTS_EVENT_NODE_ONLINE` you will be allowed to join or leave networks. You must authorize the node ID provided by the this callback event to join your network. This can be done manually or via our Web API. Note however that if you are using an Ad-hoc network, it has no controller and therefore requires no authorization.

At the end of your program or when no more network activity is anticipated, the user application can shut down the service with `zts_stop()`. However, it is safe to leave the service running in the background indefinitely as it doesn't consume much memory or CPU while at idle. `zts_stop()` is a non-blocking call and will itself issue a series of events indicating that various aspects of the ZeroTier service have successfully shut down.

It is worth noting that while `zts_stop()` will stop the service, but the user-space network stack will continue operating in a headless hibernation mode. This is intended behavior due to the fact that the network stack we've chosen doesn't currently support the notion of shutdown since it was initially designed for embedded applications that are simply switched off. If you do need a way to shut everything down and free all resources you can call `zts_free()`, but please note that calling this function will prevent all subsequent `zts_start()` calls from succeeding and will require a full application restart if you want to run the service again. The events `ZTS_EVENT_NODE_ONLINE` and `ZTS_EVENT_NODE_OFFLINE` can be seen periodically throughout the lifetime of your application depending on the reliability of your underlying network link, these events are lagging indicators and are typically only triggered every thirty (30) seconds.

Lastly, the function `zts_restart()` is provided as a way to restart the ZeroTier service along with all of its virtual interfaces. The network stack will remain online and undisturbed during this call. Note that this call will temporarily block until the service has fully shut down, then will return and you may then watch for the appropriate startup callbacks mentioned above.

Joining a network

Joining a ZeroTier virtual network is as easy as calling `zts_join(uint64_t networkId)`. Similarly there is a `zts_leave(uint64_t networkId)`. Note that `zts_start()` must be called and a `ZTS_EVENT_NODE_ONLINE` event must be received before these calls will succeed. After calling `zts_join()` any one of the following events may be generated:

```
ZTS_EVENT_NETWORK_NOT_FOUND
ZTS_EVENT_NETWORK_CLIENT_TOO_OLD
ZTS_EVENT_NETWORK_REQUESTING_CONFIG
ZTS_EVENT_NETWORK_OK
ZTS_EVENT_NETWORK_ACCESS_DENIED
ZTS_EVENT_NETWORK_READY_IP4
ZTS_EVENT_NETWORK_READY_IP6
ZTS_EVENT_NETWORK_DOWN
```

`ZTS_EVENT_NETWORK_READY_IP4` and `ZTS_EVENT_NETWORK_READY_IP6` are combinations of a few different events. They signal that the network was found, joined successfully, an IP address was assigned and the network stack's interface is ready to process traffic of the indicated type. After this point you should be able to communicate with peers on the network.

Connecting and communicating with peers

Creating a standard socket connection generally works the same as it would using an ordinary socket interface, however with ZeroTier there is a subtle difference in how connections are established which may cause confusion. Since ZeroTier employs transport-triggered link provisioning a direct connection between peers will not exist until contact has been attempted by at least one peer. During this time before a direct link is available traffic will be handled via our free relay service. The provisioning of this direct link usually only takes a couple of seconds but it is important to understand that if you attempt something like s `zts_connect(...)` call during this time it may fail due to packet loss. Therefore it is advised to repeatedly call `zts_connect(...)` until it succeeds and to wait to send additional traffic until `ZTS_EVENT_PEER_P2P` has been received for the peer you are attempting to communicate with. All of the above is optional, but it will improve your experience.

`tl;dr: Try a few times and wait a few seconds`

As a mitigation for the above behavior, ZeroTier will by default cache details about how to contact a peer in the `peers.d` subdirectory of the config path you passed to `zts_start(...)`. In scenarios where paths do not often change, this can almost completely eliminate the issue and will make connections nearly instantaneous. If however you do not wish to cache these details you can disable it via `zts_set_peer_caching(false)`.

One can use `zts_get_peer_status(uint64_t peerId)` to query the current reachability state of another peer. This function will actually **return** value of the previously observed callback event for the given peer, plus an additional possible value `ZTS_EVENT_PEER_UNREACHABLE` if no known path exists between the calling node and the remote node.

Event handling

As mentioned in previous sections, the control API works by use of non-blocking calls and the generation of a few dozen different event types. Depending on the type of event there may be additional contextual information attached to the `zts_callback_msg` object that you can use. This contextual information will be housed in one of the following structures which are defined in `include/ZeroTier.h`:

```
struct zts_callback_msg
{
    int eventCode;
    struct zts_node_details *node;
    struct zts_network_details *network;
    struct zts_netif_details *netif;
    struct zts_virtual_network_route *route;
    struct zts_physical_path *path;
    struct zts_peer_details *peer;
    struct zts_addr_details *addr;
};
```

Here's an example of a callback function:

```
void myZeroTierEventCallback(struct zts_callback_msg *msg)
{
    if (msg->eventCode == ZTS_EVENT_NODE_ONLINE) {
        printf("ZTS_EVENT_NODE_ONLINE, node=%llx\n", msg->node->address);
        // You can join networks now!
    }
}
```

In this callback function you can perform additional non-blocking API calls or other work. While not returning control to the service isn't forbidden (the event messages are generated by a separate thread) it is recommended that you return control as soon as possible as not returning will prevent the user application from receiving additional callback event messages which may be time-sensitive.

A typical ordering of messages may look like the following:

```
ZTS_EVENT_NETIF_UP --- network=a09acf023be465c1, mac=73b7abcfc207, mtu=10000
ZTS_EVENT_ADDR_NEW_IP4 --- addr=11.7.7.184 (on network=a09acf023be465c1)
ZTS_EVENT_ADDR_NEW_IP6 --- addr=fda0:9acf:233:e4b0:7099:9309:4c9b:c3c7
ZTS_EVENT_NODE_ONLINE, node=c4c7ba3cf
ZTS_EVENT_NETWORK_READY_IP4 --- network=a09acf023be465c1
ZTS_EVENT_NETWORK_READY_IP6 --- network=a09acf023be465c1
ZTS_EVENT_PEER_P2P --- node=74d0f5e89d
ZTS_EVENT_PEER_P2P --- node=9d219039f3
ZTS_EVENT_PEER_P2P --- node=a09acf0233
```

Node Events

These events pertain to the state of the current node. This message type will arrive with a `zts_node_details` object accessible via `msg->node`. Additionally, one can query the status of the node with `zts_get_node_status()`, this will **return** the status as an integer value only.

```
ZTS_EVENT_NODE_OFFLINE
ZTS_EVENT_NODE_ONLINE
ZTS_EVENT_NODE_DOWN
ZTS_EVENT_NODE_IDENTITY_COLLISION
ZTS_EVENT_NODE_UNRECOVERABLE_ERROR
ZTS_EVENT_NODE_NORMAL_TERMINATION
```

Network Events

These events pertain to the state of the indicated network. This event type will arrive with a `zts_network_details` object accessible via `msg->network`. If for example you want to know the number of assigned routes for your network you can use `msg->network->num_routes`. Similarly for the MTU, use `msg->network->mtu`. Additionally, one can query the status of the network with `zts_get_network_status(uint64_t networkId)`, this will **return** the status as an integer value only.

```
ZTS_EVENT_NETWORK_NOT_FOUND
ZTS_EVENT_NETWORK_CLIENT_TOO_OLD
ZTS_EVENT_NETWORK_REQUESTING_CONFIG
ZTS_EVENT_NETWORK_OK
ZTS_EVENT_NETWORK_ACCESS_DENIED
ZTS_EVENT_NETWORK_READY_IP4
ZTS_EVENT_NETWORK_READY_IP6
ZTS_EVENT_NETWORK_DOWN
```

Peer Events

These events are triggered when the reachability status of a peer has changed, this can happen at any time. This event type will arrive with a `zts_peer_details` object for additional context. Additionally, one can query the status of the network with `zts_get_peer_status(uint64_t peerId)`, this will **return** the status as an integer value only.

```
ZTS_EVENT_PEER_P2P
ZTS_EVENT_PEER_RELAY
ZTS_EVENT_PEER_UNREACHABLE
```

Path Events

These events are triggered when a direct path to a peer has been discovered or is now considered too old to be used. You will see these in conjunction with peer events. This event type will arrive with a `zts_physical_path` object for additional context.

```
ZTS_EVENT_PATH_DISCOVERED
ZTS_EVENT_PATH_ALIVE
ZTS_EVENT_PATH_DEAD
```

Route Events

This event type will arrive with a `zts_virtual_network_route` object for additional context.

```
ZTS_EVENT_ROUTE_ADDED
ZTS_EVENT_ROUTE_REMOVED
```

Address Events

These events are triggered when new addresses are assigned to the node on a particular virtual network. This event type will arrive with a `zts_addr_details` object for additional context.

```
ZTS_EVENT_ADDR_ADDED_IP4
ZTS_EVENT_ADDR_REMOVED_IP4
ZTS_EVENT_ADDR_ADDED_IP6
ZTS_EVENT_ADDR_REMOVED_IP6
```

Error handling

Calling a `zts_*` function will result in one of the following return codes. Only when `ZTS_ERR` is returned will `zts_errno` be set. Its values closely mirror those used in standard socket interfaces and are defined in `./ext/lwip/src/include/lwip/errno.h`.

- `ZTS_ERR_OK`: No error
- `ZTS_ERR`: Error (see `zts_errno`for more information)
- `ZTS_ERR_INVALID_ARG`: An argument provided is invalid.
- `ZTS_ERR_SERVICE`: ZT is not yet initialized. Try again.
- `ZTS_ERR_INVALID_OP`: Operation is not permitted (Doesn't make sense in this state).
- `ZTS_ERR_NO_RESULT`: Call succeeded but no result was available. Not necessarily an error.
- `ZTS_ERR_GENERAL`: General internal failure. Consider filing a bug report.

NOTE: For Android/Java (or similar) which use JNI, the socket API's error codes are negative values encoded in the return values of function calls NOTE: For protocol-level errors (such as dropped packets) or internal network stack errors, see the section `Statistics`

Common pitfalls

- If you have started a node but have not received a `ZTS_EVENT_NODE_ONLINE`:
    - You may need to view our Router Config Tips knowledgebase article. Sometimes this is due to firewall/NAT settings.
- If you have received a `ZTS_EVENT_NODE_ONLINE` event and attempted to join a network but do not see your node ID in the network panel on my.zerotier.com after some time:
    - You may have typed in your network ID incorrectly.
    - Used an improper integer representation for your network ID (e.g. `int` instead of `uint64_t`).
- If you are unable to reliably connect to peers:
    - You should first read the section on **Connecting and communicating with peers**.
    - If the previous step doesn't help move onto our knowledgebase article Router Config Tips. Sometimes this can be a transport-triggered link issue, and sometimes it can be a firewall/NAT issue.
- API calls seem to fail in nonsensical ways and you're tearing your hair out:
    - Be sure to read and understand the **API compatibility with host OS** section.
    - See the **Debugging** section for more advice.

API compatibility with host OS

Since libzt re-implements a socket interface likely very similar to your host OS's own interface it may be tempting to mix and match host OS structures and functions with those of libzt. This may work on occasion, but you are tempting fate, here are a few important guidelines:

If you are calling a `zts_*` function, use the appropriate `ZTS_*` constants:

```
zts_socket(ZTS_AF_INET6, ZTS_SOCK_DGRAM, 0); (CORRECT)
zts_socket(AF_INET6, SOCK_DGRAM, 0);         (INCORRECT)
```

If you are calling a `zts_*` function, use the appropriate `zts_*` structure:

```
struct zts_sockaddr_in in4;  <------ Note the zts_ prefix
    ...
zts_bind(fd, (struct sockaddr *)&in4, sizeof(struct zts_sockaddr_in)) < 0)
```

If you are calling a host OS function, use your host OS's constants (and structures!):

```
inet_ntop(AF_INET6, &(in6->sin6_addr), ...);     (CORRECT)
inet_ntop(ZTS_AF_INET6, &(in6->sin6_addr), ...); (INCORRECT)
```

If you are calling a host OS function but passing a `zts_*` structure, this can work sometimes but you should take care to pass the correct host OS constants:

```
struct zts_sockaddr_in6 in6;
    ...
inet_ntop(AF_INET6, &(in6->sin6_addr), dstStr, INET6_ADDRSTRLEN);
```

Thread model

The control API for `libzt` is thread safe and can be called at any time from any thread. There is a single internal lock guarding access to this API. The socket API is similar in this regard. Callback events are generated by a separate thread and are independent from the rest of the API's internal locking mechanism. Not returning from a callback event won't impact the rest of the API but it will prevent your application from receiving future events so it is in your application's best interest to perform as little work as possible in the callback function and promptly return control back to ZeroTier.

Note: Internally, `libzt` will spawn a number of threads for various purposes: a thread for the core service, a thread for the network stack, a low priority thread to process callback events, and a thread for each network joined. The vast majority of work is performed by the core service and stack threads.

Debugging

If you're experiencing odd behavior or something that looks like a bug I would suggest first reading and understanding the following sections:

- **Common pitfalls**
- **API compatibility with host OS**
- **Thread model**

If the information in those sections hasn't helped, there are a couple of ways to get debug traces out of various parts of the library.

1. Build the library in debug mode with `make host_debug`. This will prevent the stripping of debug symbols from the library and will enable basic output traces from libzt.

2. If you believe your problem is in the network stack you can manually enable debug traces for individual modules in `src/lwipopts.h`. Toggle the `*_DEBUG` types from `LWIP_DBG_OFF` to `LWIP_DBG_ON`. And then rebuild. This will come with a significant performance cost.

3. Enabling network stack statistics. This is useful if you want to monitor the stack's receipt and handling of traffic as well as internal things like memory allocations and cache hits. Protocol and service statistics are available in debug builds of `libzt`. These statistics are detailed fully in the section of `include/ZeroTier.h` that is guarded by `LWIP_STATS`. The protocol constants are defined in `include/ZeroTierConstants.h`. An example usage is as follows:

```
struct zts_stats_proto stats;

// Get count of received pings
if (zts_get_protocol_stats(ZTS_STATS_PROTOCOL_ICMP, &stats) == ZTS_ERR_OK) {
    printf("icmp.recv=%d\n", stats.recv);
}

// Get count of dropped TCP packets
if (zts_get_protocol_stats(ZTS_STATS_PROTOCOL_TCP, &stats) == ZTS_ERR_OK) {
    printf("tcp.drop=%d\n", stats.drop);
}
```

4. There are a series of additional events which can signal whether the network stack or its virtual network interfaces have been set up properly:

Network stack events - These signal whether the userspace networking stack was brought up successfully. You can ignore these in most cases. This event type will arrive with no additional contextual information.

```
ZTS_EVENT_STACK_UP
ZTS_EVENT_STACK_DOWN
```

Netif events - These signal whether the userspace networking stack was brought up successfully. You can ignore these in most cases. This event type will arrive with a `zts_netif_details` object for additional context.

```
ZTS_EVENT_NETIF_UP
ZTS_EVENT_NETIF_DOWN
ZTS_EVENT_NETIF_REMOVED
ZTS_EVENT_NETIF_LINK_UP
ZTS_EVENT_NETIF_LINK_DOWN
```

Network controller mode

The library form of ZeroTier can act as a network controller and in `libzt` this is controlled via the `zts_controller_*` API calls specified in `include/ZeroTier.h`. Currently controller mode is not available in the `iOS` and `macOS` framework builds.

Multipath

The multipath features available in ZeroTier haven't yet been exposed via the `libzt` API but this is coming in the version `2.X` series.

Licensing

ZeroTier is licensed under the BSL version 1.1. See LICENSE.txt and the ZeroTier pricing page for details. ZeroTier is free to use internally in businesses and academic institutions and for non-commercial purposes. Certain types of commercial use such as building closed-source apps and devices based on ZeroTier or offering ZeroTier network controllers and network management as a SaaS service require a commercial license.

A small amount of third party code is also included in ZeroTier and is not subject to our BSL license. See AUTHORS.md for a list of third party code, where it is included, and the licenses that apply to it. All of the third party code in ZeroTier is liberally licensed (MIT, BSD, Apache, public domain, etc.). If you want a commercial license to use the ZeroTier SDK in your product contact us directly via contact@zerotier.com