UDT Tutorial

Configure UDT Options

UDT 选项配置

Options of UDT are read and set through getsockopt and setsockopt methods. Before modifying any option, bear in mind that it is NOT required that you modify the default options. If the application has sound performance with the default options, just use the default configurations.

通过 getsockopt 和 setsockopt 函数读取与设置 UDT 选项。在修改任何默认选项之前，请牢记，默认选项的修改不是必须的，如果应用程序在默认选项下可以工作，就使用默认选项。

UDT_MSS is used to configure the packet size. In most situations, the optimal UDT packet size is the network MTU size. The default value is 1500 bytes. A UDT connection will choose the smaller value of the MSS between the two peer sides. For example, if you want to set 9000-byte MSS, you have to set this option at both sides, and one of the value has to be exactly equal to 9000, and the other must not be less than 9000.

UDT_MSS 用于配置数据包大小。在大多数情况下，最优的 UDT 数据包大小是网络 MTU 大小。默认值为 1500 字节。UDT 连接将选择两个对端之间较小的 MSS 值。 例如，如果要设置 9000 字节 MSS，则必须在两侧设置此选项，其中一个值必须等于 9000，而另一个必须不小于 9000。

UDT uses a different semantics of synchronization mode from traditional sockets. It can set the sending and receiving synchronization independently, which allows more flexibility. However, UDT does not allow non-blocking operation on connection setup and close. The synchronization mode of sending and receiving can be set on UDT_SNDSYN and UDT_RCVSYN, respectively.

UDT 使用与传统套接字不同的同步模式语义。它可以独立设置发送和接收同步，这允许更多的灵活性。但是，UDT 不允许在连接建立和关闭时进行非阻塞操作。 发送和接收的同步模式可以分别在 UDT_SNDSYN 和 UDT_RCVSYN 上设置。

The UDT buffer size is (UDT_SNDBUF and UDT_RCVBUF) used to limit the size of temporary storage of sending/receiving data. The buffer size is only a limit and memory is allocated upon necessary. Generally, larger buffer (but not so large that the physical memory is used up) is better. For good performance the buffer sizes for both sides should be at least *Bandwidth*RTT*.

UDT 缓冲区大小为（UDT_SNDBUF 和 UDT_RCVBUF），用于限制发送/接收数据的临时存储大小。 缓冲区大小只是一个限制，必要时分配内存。 通常情况下，较大的缓冲区更好（但不必太大，以免物理内存用尽）。为了表现良好，双方的缓冲区大小应至少为带宽*RTT。

UDT uses UDP as the data channel, so the UDP buffer size affects the performance. Again, a larger value is generally better, but the effects become smaller and disappear as the buffer size increases. Generally, the sending buffer size can be a small value, because it does not limit the packet sending much but a large value may increase the end-to-end delay.

UDT 使用 UDP 作为数据通道，因此 UDP 缓冲区大小会影响性能。再次，更大的值通常更好，但是效果变得更小并且随着缓冲器大小的增加而消失。通常，发送缓冲区大小可以是一个小的值，因为它不会限制数据包的发送，但是大的值可能会增加端到端的延迟。

UDT_FC is actually an internal parameter and you should set it to not less than UDT_RCVBUF/UDT_MSS. The default value is relatively large, therefore unless you set a very large receiver buffer, you do not need to change this option.

UDT_FC 实际上是一个内部参数，您应该将其设置为不小于 UDT_RCVBUF / UDT_MSS。 默认值相对较大，因此除非设置了非常大的接收缓冲区，否则不需要更改此选项。

UDT_LINGER is similar to the SO_LINGER option on the regular sockets. It allows the UDT socket continue to sent out data in the sending buffer when close is called.

UDT_LINGER 类似于常规套接字上的 SO_LINGER 选项。它允许 UDT 套接字在关闭调用时继续发送发送缓冲区中的数据。

UDT_RENDEZVOUS is used to enable rendezvous connection setup. When rendezvous mode is enabled, a UDT socket cannot call listen or accept; instead, in order to set up a rendezvous connection, both the peer sides must call connect at approximately the same time. This is useful in traversing a firewall.

UDT_RENDEZVOUS 用于启用会合连接设置。当会合模式启用时，UDT 套接字不能调用 listen 或 accept; 相反，为了建立会合连接，对方必须在大约相同的时间调用连接，主要用于防火墙穿透。

UDT_SNDTIMEO and UDT_RCVTIMEO are similar to SO_SNDTIMEO and SO_RCVTIMEO, respectively. They are used to set a timeout value for packet sending and receiving.

UDT_SNDTIMEO 和 UDT_RCVTIMEO 分别与 SO_SNDTIMEO 和 SO_RCVTIMEO 类似。它们用于设置数据包发送和接收的超时值。

UDT_REUSEADDR allows applications to decide whether to share a UDP port with other UDT connections. By default this option is true, which means all UDT connections that are bind to 0 will try to reuse any existing UDP socket. In addition, multiple UDT connections can bind to the same port number other than 0. If UDT_REUSEADDR is set to false, an exclusive UDP port will be assign to this UDT socket. There are a few situations when UDT_REUSEADDR should be set to false. First, two UDT sockets cannot listen on the same port number, so either the second UDT socket is explicitly bound to a different port, or UDT_REUSEADDR is set to false for this UDT socket. Second, a UDT socket bound to a specific port number cannot connect to the other UDT socket bound to the same port on the same IP address.

UDT_REUSEADDR 允许应用程序决定是否与其他 UDT 连接共享 UDP 端口。默认情况下，此选项为 true，这意味着绑定到 0 的所有 UDT 连接将尝试重用任何现有的 UDP 套接字。此外，多个 UDT 连接可以绑定到除了 0 以外的相同端口号。如果 UDT_REUSEADDR 设置为 false，则会将独占 UDP 端口分配给此 UDT 套接字。UDT_REUSEADDR 应设置为 false 时有几种情况。首先，两个 UDT 套接字不能侦听相同的端口号，因此第二个 UDT 套接字显式绑定到另一个端口，或 UDT 套接字的 UDT_REUSEADDR 设置为 false。第二，绑定到特定端口号的 UDT 套接字不能连接到同一 IP 地址上绑定到同一端口的其他 UDT 套接字。

Transfering Data using UDT

使用 UDT 传输数据

This section describes using UDT to transfer data in streaming mode. This is exactly the same as using traditional BSD socket.

本节介绍如何使用 UDT 以流式传输数据。这与使用传统的 BSD 套接字完全相同。

In streaming mode, neither a send or a recv call can guarantee that all data are sent or received in one call, because there is no boundary information in the data stream. Application should use loops for both sending and receiving.

在流式传输模式下，由于在数据流中没有边界信息，所以发送或接收呼叫都不能保证在一个呼叫中发送或接收所有数据。 应用程序应该使用循环来发送和接收。

UDT supports both blocking and non-blocking mode. The above example demonstrated the blocking mode. In non-blocking mode, UDT::send and UDT::recv will return immediately if there is no buffer available. Usually, non-blocking calls are used together with accept.

UDT 支持阻塞和非阻塞模式。上面的例子展示了阻塞模式。在非阻塞模式下，如果没有缓冲区可用，UDT::send, UDT::recv 将立即返回。通常，非阻塞调用与 accept 一起使用。

UDT also supports timed blocking IO with UDT_SNDTIMEO and UDT_RCVTIMEO. This is in the middle between complete blocking and complete non-blocking calls. Timed IO will block the sending or receiving call for a limited period. This is sometimes useful if the application does not know if and when the peer side will send a message.

UDT 还支持使用 UDT_SNDTIMEO 和 UDT_RCVTIMEO 定时阻塞 IO。这是在完全阻塞和完全非阻塞之间的中间调用。 定时 IO 将在有限的时间段内阻塞发送或接收调用。 如果应用程序不知道对端何时发送消息时，可以使用定时阻塞 IO。

Messaging with Partial Reliability

部分可靠消息模式

When a UDT socket is created as SOCK_DGRAM type, UDT will send and receive data as messages. The boundary of the message is preserved and the message is delivered as a whole unit. Sending or receiving messages do not need a loop; a message will be either completely delivered or not delivered at all. However, at the receiver side, if the user buffer is shorter than the message length, only part of the message will be copied into the user buffer while the message will still be discarded.

当 UDT 套接字创建为 SOCK_DGRAM 类型时，UDT 将作为消息发送和接收数据。 消息的边界被保留，消息作为一个整体传送。发送或接收消息不需要循环； 一条消息将完全交付或完全没有交付。然而，在接收方，如果用户缓冲区短于消息长度，则只有部分消息将被复制到用户缓冲区，而消息仍将被丢弃。

At the sender side, applications can specify two options for every message. The first is the life time (TTL) of a message. The default value is infinite, which means that the message will always be delivered. If the value is a positive one, UDT will discard the message if it cannot be delivered by the life time expires. The second is the order of the message. An in-order message means that this message will not be delivered unless all the messages prior to it are either delivered or discarded.

在发送方，应用程序可以为每个消息指定两个选项。第一个是消息的生命周期（TTL）。默认值为无穷大，这意味着消息将始终被传递。如果该值为一个值，则 UDT 将丢弃该消息，如果它在生命周期到期后无法传递。第二个是消息的顺序。有序消息就是说除非该消息之前的所有消息传输完成或者被丢弃，否则该消息不会被传输。

Synchronization modes (blocking vs. non-blocking) are also applied to SOCK_DGRAM sockets, so does not other UDT mechanisms including but limited to congestion control, flow control, and connection maintenance. Finally, note that UDT SOCK_DGRAM socket is also connection oriented. A UDT connection can only be set up between the same socket types.

同步模式（阻塞与非阻塞）也适用于 SOCK_DGRAM 套接字，因此不包括但不限于拥塞控制，流量控制和连接维护等其他 UDT 机制。最后，请注意，UDT SOCK_DGRAM 套接字也是面向连接的。UDT 连接只能在相同的套接字类型之间建立。

File Transfer using UDT

UDT 文件传输

While you can always use regular UDT::send and UDT::recv to transfer a file, UDT provides a more convinient and op-timized way for file transfer. An application can use UDT::sendfile and UDT::recvfile directly. In addition, file transfer IO API and regular data IO API are orthogonal. E.g., the data stream sent out by UDT::sendfile does not necessarily require UDT::recvfile to accept.

尽管总是可以使用常规 UDT::send 和 UDT::recv 传输文件，但 UDT 为文件传输提供了更方便和最优化的方式。 应用程序可以直接使用 UDT::sendfile 和 UDT::recvfile。此外，文件传 IO 的 API 和常规数据 IO 的 API 是正交的。 例如，UDT::sendfile 发出的数据流不一定要求 UDT::recvfile 接受。

The sendfile and recvfile methods are blocking call and are not affected by UDT_SNDSYN, UDT_RCVSYN, UDT_SBDTIMEO, or UDT_RCVTIMEO. They always complete the call with the specified size parameter for sending or receiving unless errors occur.

sendfile 和 recvfile 方法是阻塞调用，不受 UDT_SNDSYN，UDT_RCVSYN，UDT_SBDTIMEO 或 UDT_RCVTIMEO 的影响。 除非发生错误，否则他们总是使用指定的大小参数来完成发送或接收调用。

Firewall Traversing with UDT

UDT 防火墙穿透

While UDT was originally written for extremely high speed data transfer, there are many other potential benefits from this reliable UDP-based library. One particular usage is to setup reliable connections between machines behind firewalls. To meet this requirement, UDT has added the rendezvous connection setup support.

虽然 UDT 最初是为高速数据传输而编写的，但是这种可靠的基于 UDP 的库还有许多其他的潜在优势。一种特殊的用途是在防火墙之

后的机器之间建立可靠的连接。为了满足这一要求，UDT 增加了会合连接设置支持。

Traditional BSD socket setup process requires explicit server side and client side. To punch NAT firewalls, a common method is to use the SO_REUSEADDR socket option to open two sockets bound to the same port, one listens and the other connects. UDT provides the more convenient rendezvous connection setup, in which there is no server or client, and two users can connect to each other directly.

传统的 BSD 套接字建立连接过程需要显式的服务器端和客户端。要打洞 NAT 防火墙，常用的方法是使用 SO_REUSEADDR 套接字选项来打开绑定到同一个端口的两个套接字，一个侦听，另一个连接。UDT 提供了更方便的会合连接设置，其中没有服务器或客户端，两个用户可以直接相互连接。

With UDT, all sockets within one process can be bound to the same UDP port (but at most one listening socket on the same port is allowed). This is also helpful for system administrators to open a specific UDP port for all UDT traffic.

使用 UDT，一个进程内的所有套接字都可以绑定到相同的 UDP 端口（但同一个端口上最多允许一个侦听套接字）。这也有助于系统管理员为所有 UDT 流量打开特定的 UDP 端口。

In addition, UDT also allows to bind on an existing UDP socket. This is useful in two situations. First, sometimes the application must send packet to a name server in order to obtain its address (for example, this is true when behind an NAT firewall). Users may create a UDP socket and send some UDP packets to the name server to obtain the binding address. Then the UDP socket can be used directly for UDT (see bind) so that the application does not need to close the UDP socket and open a new UDT socket on the same address again.

此外，UDT 还允许绑定在现有的 UDP 套接字上。 这在以下两种情形下是有用的：首先，有时应用程序必须向指定的服务器发送数据包以获取其地址（例如，在 NAT 防火墙后面是这样）。 用户可以创建一个 UDP 套接字，并向指定的服务器发送一些 UDP 数据包，以获取绑定地址。 然后，UDP 套接字可以直接用于 UDT（请参阅 bind），以便应用程序不需要关闭 UDP 套接字并再次打开同一个地址上新的 UDT 套接字。

Second, some firewalls working on local system may change the port mapping or close the "hole" is the punching UDP socket is closed, thus a new UDT socket on the same address will not be able to traverse the firewall. In this situation, binding the UDT socket on the existing UDP socket is not only convenient but necessary.

第二，一些在本地系统上工作的防火墙可能会改变端口映射或关闭“洞”，即打洞 UDP 套接字被关闭，因此在相同地址上的新 UDT 套接字将无法穿透防火墙。 在这种情况下，绑定现有 UDP 套接字上的 UDT 套接字不但方便而且必要。

Error Handling

错误处理

All UDT API will return an error upon a failed operation. Particularly, UDT defines UDT::INVALID_SOCK and UDT::ERROR as error returned values. Application should check the return value against these two constants (several routine return false as error value).

所有 UDT API 将在调用失败时返回错误。特别地，UDT 将 UDT :: INVALID_SOCK 和 UDT :: ERROR 定义为错误返回值。应用程序应该根据这两个常量检查返回值（有几个函数返回 false 作为错误值）。

On error, getlasterror can be used to retrieve the error information. In fact, the function returns the latest error occurred in the thread where the function is called. getlasterror returns an ERRORINFO structure, it contains both the error code and special text error message. Two helper functions of getErrorCode and getErrorMessage can be used to read these information.

出错时，getlasterror 函数可用于检索错误信息。 实际上，该函数返回在函数调用线程中发生的最新错误。getlasterror 返回一个 ERRORINFO 结构，它同时包含错误代码和特殊文本错误消息。getErrorCode 和 getErrorMessage 的两个辅助函数可用于读取这些信息。

The UDT error information is thread local (that is, an error in another thread will not affect the error information in the current thread). The returned value is a reference to the UDT internal error structure.

UDT 错误信息是线程本地的（也就是说，另一个线程中的错误不会影响当前线程中的错误信息）。返回值是对 UDT 内部错误结构的引用。

Note that a successful call will NOT clear the error. Therefore, applications should use the return value of a UDT API to check the result of a UDT call. getlasterror only provides detailed information when necessary. However, application can use getlasterror().clear() to clear the previously logged error if needed.

请注意，成功的调用并不会清除错误。 因此，应用程序应使用 UDT API 的返回值来检查 UDT 调用的结果。 getlasterror 仅在必要时提供详细信息。 但是，如果需要，应用程序可以使用 getlasterror().clear()来清除以前记录的错误。

The UDT error code only reflects the operation error of the UDT socket level. Applications can still read the system level error (e.g., errno in Linux, GetLastError in Windows) to read more specific error information. However, the error message obtained by getErrorMessage contains information of both the UDT level error and the system level error.

UDT 错误代码仅反映 UDT 套接字级别的操作错误。应用程序仍然可以读取系统级错误（例如，Linux 中的 errno，Windows 中的 GetLastError）以获得更具体的错误信息。但是，getErrorMessage 获取的错误消息包含 UDT 级错误和系统级错误的信息。

User-defined Congestion Control Algorithm

用户自定义拥塞控制算法

You can add your own congestion control algorithm into UDT. It is as simple as to define several callback functions that will be triggered on certain events, e.g, when an ACK is received.

你可以将自己的拥塞控制算法添加到 UDT 中，简单的说，只需要定义几个在某些事件上被触发的回调函数，例如当接收到 ACK 时的回调函数。

All the congestion control callback functions are collected in a C++ class CCC. You have to inherit this class to define your own congestion control algorithm. That is, UDT/CCC uses an objectoriented design. CCC in defined in ccc.h, which you have to include in your files in order to enable this feature.

所有拥塞控制回调函数集中在 C++类 CCC 中。你必须继承这个类来定义自己的拥塞控制算法。也就是说，UDT/CCC 使用面向对象的设计。CCC 定义在 ccc.h 中，你必须将其包含在你的文件中才能启用此功能。

The CCC class contains two control variables: m_dPktSndPeriod, and m_dCWndSize. m_dPktSndPeriod is a double float number representing the packet sending period (as to be used in rate control), in microseconds. m_dCWndSize is a double float number representing the size of the congestion window (cwnd), in number of packets. The congestion control algorithm will need to update at least one of them. For example, for pure window based approach, m_dPktSndPeriod should always be zero.

CCC 类包含两个控制变量：m_dPktSndPeriod 和 m_dCWndSize。m_dPktSndPeriod 是表示数据包发送周期（用于速率控制）的双浮点数，以微秒为单位。m_dCWndSize 是一个双浮点数，表示拥塞窗口（cwnd）的大小（以数据包为单位）。拥塞控制算法将需要更新其中至少一个。例如，对于基于纯窗口的方法，m_dPktSndPeriod 应始终为零。

The fast way to learn CCC is to use the examples in ./app/cc.h. The file cc.h also includes many more advanced control mechanisms that your control classes can be derived from. For example, if you are designing a new TCP variant, you can implement the new control class directly from CTCP.

学习 CCC 的快速方法是使用./app/cc.h 中的示例。文件 cc.h 还包括许多更高级的控制机制，自定义的控制类可以从中继承。例如，

如果你正在设计新的 TCP 变体，则可以直接从 CTCP 实现新的控制类。

Here we demonstrate the usage of UDT/CCC by writing a reliable UDP blast control mechanism.

这里我们通过编写可靠的 UDP 流控制机制来演示 UDT/CCC 的用法。