UDT Based Application Performance For High Speed Data Transfer

Joshua Miller
jsmiller@uchicago.edu
The University of Chicago

Supercomputing 1013

Abstract

TCP is the most widely used reliable network transport protocol. However, over high performance, wide area networks, TCP has been show to reach a bottleneck before UDP. UDT (UDP Based Data transfer) is a reliable application level protocol for transferring bulk data over wide area networks. In order to make UDT more usable, LAC (the Laboratory for Advanced Computing) has created UDR, a wrapper for rsync that enables rsync to use UDT. However, rsync was not designed to make use of the performance gains introduced by UDT, and therefore the increase in through put is significant but not optimal. Therefore the first major goal of this project was to explore strategies to increase the performance gains of UDR.

If the data transfer is to be encrypted, an encryption package is needed that can match the throughput of the UDT data transfer. The second major goal of this project was to create a threaded encryption system using OpenSSL's EVP cipher library.

The third major goal of this project was to create a new application to eliminate the overhead introduced into UDR by rsync by implementing the functionality of netcat over UDT.

This material is based upon work supported by the National Science Foundation under Grants No. CISE-1127316 and OISE-1129076.

UDR

The first attempt at increasing the throughput of UDR was to recreate it as a wrapper for scp as well as rsync. The motivation for this lies in the fact that preliminary tests showed a higher transfer rate for scp than rsync. However, once included, scp did not demonstrate a significantly higher throughput over UDT than did rsync (1.22 Gbps versus 1.13 Gbps respectively).

In order to increase the performance of an encrypted UDR transfer, a threaded implementation of OpenSSL was introduced (see section Encryption below for details). Machines with processors not capable of encrypting at the data transfer rate of UDR, are now able to match the data transfer rate of UDR if multiple cores are available.

While working with UDT, it was noted that optimal performance was achieved with a buffer size of 64 MB. However, rsync operates with a buffer size of 32768 bytes. An attempt was made to buffer data until the optimal UDT block size was achieved, while sending immediately if an rsync communication (not data) block, was received. This strategy, however, did not increase UDR throughput because additional delay was introduced in the read loop.

A future goal is to create a patch to rsync to adjust the buffer size and hopefully the overall performance.

UDTCAT

Netcat is a simple method of transferring data over a network, be it a pipe, a file redirect, or simply a TTY input. The original intention for UDTCAT was to replication the functionality of netcat over UDT. It has since been expanded to included encryption and authentication. Therefore, UDTCAT is a secure, bi-directional method for writing data across UDT.

Each UDTCAT process consists of a receiving and sending thread. Each receiving and sending thread is the parent to N encryption threads (set at runtime). UDTCAT is not encrypted by default. Specifying a key file or a key in line enables encryption. When encryption is enabled, each receiving

thread requests that the peer's sending thread sign a block of randomly generated bytes by encrypting it. If the block, once decrypted by the receiving thread, matches the original random block then data exchange proceeds as normal.

The most straight-forward use of UDTCAT is to transfer a file. The two commands below transfer the input_file to the output_file over localhost using 16 encryption threads and the contents of key_file as the encryption key.

By avoiding the over head of rsync, UDTCAT is able to achieve unencrypted and encrypted transfers of 7.5 and 6.4 Gbps respectively.

```
$ uc -n16 -f key_file localhost  \
                9000 < input_file
$ uc -n16 -f key_file -l \
                9000 > output_file
```

For performance reasons it is important that the files are redirected to stdin and not piped in using cat. A significant decrease in speed occurs because the UDT buffer size is then regulated by cat.

Another use of UDTCAT reflects its dynamic ability to pipe data over UDT. It is possible to bind a server, such as an rsync server to a TCP port and create a two way pipe between a netcat process bound to the TCP port and a UDTCAT process bound to a remote peer. By doing this, the I/O of the server can be tunneled across UDT.

Threaded Encryption

There is very limited literature on the support of threading in OpenSSL. A method of threaded encryption using a single EVP cipher was not found, even with the required OpenSSL call-back functions. Therefore a system to use multiple threads had to be designed to match UDT throughput. The system utilizes N identical OpenSSL EVP ctx ciphers for each encryption and decryption process. Because the ciphers update themselves after each call to OpenSSL's encrypt and decrypt functions, the data must be aligned on the encrypting and decrypting sides. In order to achieve this alignment, both UDTCAT and UDR add an initial length header of 4 bytes to each UDT block. This block is divided evenly among the decryption threads on the receiving side.

Preliminary tests showed that the overhead from creating a thread during each round of encryption was costly enough to see a noticeable decrease in performance. Therefore, the threads were pooled. This eliminated the overhead, as the encryption threads now remain idle on a mutex lock until required.

The encryption package has been encapsulated such that there are necessary function calls: initialization, passing data to thread, and joining all encryption threads.

Measuring Performance

The data collected for the poster was measured using vnstat, a tool which monitors traffic over an interface. The network between the two machines had an RTT of $\approx$ 1ms, but the network tool netem was used to simulate an RTT of 75ms and a packet loss rate of .1%. These parameters were chosen due to previous testing from Chicago to Miami which demonstrated a RTT of 38 ms and a packet loss rate of .02%.

Memory to memory transfers were collected by redirecting /dev/zero to /dev/null over UDTCAT. Disk to disk transfers were collected by transferring a 50 GB file on a Seagate 400GB SSD.

To measure the rate of the encrypted transfer, single threaded tests were compared to threaded versions using 16 encryption threads per send/receive thread per process. This means that there were 32 threads spawned per process, but because the transfer was uni-directional, 16 remained idle. The processors on the machines used were the Intel Xeon CPU E5-2650 at 2.00 GHz. The performance governors of all processors were set prevent idling.

Future goals

In order to optimize UDR throughput, a to patch rsync to allow a buffer size that works better with UDT is possible.

A useful functionality for UDTCAT would be to send multiple files over the same connection. Tar presents itself as a good option to emulate or possible incorporate. Using tar with UDTCAT is currently possible, but encounters the same buffer problem as cat and rsync.

One further goal is to add the ability to establish a UDTCAT client remotely in order to make data transfer easier.