

Datagrams and Strengthening Reliability (Part 2)

The Internet offers the programmer several foundational message types to fit various programming requirements and needs. The highest level, streaming or Transaction Control Protocol (TCP) guarantees ordered, reliable data from the source to the destination program. User Datagram Protocol (UDP), or simply, datagrams offers fast, disconnected messaging. The problems with each are based on their strengths: TCP is slower than UDP, and UDP is unreliable.

This is Part 2 of a series of articles about datagrams. This article describes how to take the strengths of TCP and integrate them into UDP and create a new protocol, a reliable datagram protocol. (Please note that the Internet documents, Request For Comments or RFCs, define the Reliable Datagram Protocol or RDP. This article does not comply with that definition.)

Beefing up the Protocol

The last article described the limitations of datagrams. In particular, these are:

Unreliable – The data may not even arrive to the destination. This is more of a network limitation; and packets get dropped.

Lacks integrity – The data your program gets may be corrupted. The network transmission can garble the message.

Unordered – The sender may transmit the messages in the right order, but the receiver may get them in a different sequence. Because the message is discrete and independent, the network may use different paths between the source and destination, and the operating system makes no attempts to reorder the messages it gets.

Duplicates – The network can actually hiccup and create two or more duplicates (or *mirrors*) of the same message.

Most of these limitations are not a problem if the data does not depend on them. For example, if you write a program that sends a non-critical status to a peer host, some lost status messages are not going to cause anything to fail. However, if you want to get more out of the protocol than the typical ping, you may likely find that these limitations a real problem.

If you fixed all these problems, you basically end up with TCP. Why reinvent the wheel – especially, if the wheel works good enough? So, the idea is to select those limitations that cause the most pain. The others you can leave as “salient features” of the new protocol.

It seems that the most significant feature of datagram messaging is its discrete messaging. Getting messages in a different order seems natural to those who send mail through the post office. The other limitations are more like defects (who’s going to send a message never expecting it to be received?).

Datagrams and Strengthening Reliability

Patching the Holes

The three limitations of UDP that can cause the most pain are: unreliability, lacking integrity, and duplication. To fix each defect without losing the usefulness of datagrams (unordered), you only have to avoid reordering the messages as they arrive.

Building in reliability only requires making sure that the message eventually arrives. You can do this with either of two ways: making the receiver request any missing messages or having the sender repeat the message periodically. The first solution requires some unique identifier that the receiver uses to check off its receipt. However, this form does not lend itself well to multicast senders. So, a repeating sender can serve better – as long as the information does not change regularly.

Duplication is similar to reliability in that the client needs to account for the already arrived message and skip it. Again, this means that both the sender and the receiver must track each message with an identification code.

The last limitation, integrity, presents no real challenge to a programmer. Most modern network interface cards include a built-in checksum for the entire data message. This means that while it's possible to get scrambled data, it's not very likely. Still, you may want to add your own checksum and/or *message digest* to each message. A message digest is simply a large number that “summarizes” the entire message using some kind of hashing function. The function goes through the message byte-by-byte and calculates this hash number. Then, when the receiver gets the message, it extracts the digest, runs the function on the data, and compares the results. If they match, the message is okay.

Message digests are getting more popular and becoming far more complicated, including information about the sender, the hash function name, what services are available for interfacing the hosting peers together, etc. The real power of these summaries is in secure sockets (SSL).

The two new pieces of data that you can add in your own header to a datagram message, then, are sequence numbers (which can serve as a unique identifier) and message digests. This new header sits on top of the existing protocol's header. When the receiver gets the message, it separates the customized header from the data and verifies them. You can add whatever in your customized header, just remember that the more information you add reduces the relative amount of data that really is sent.

Putting Multicast into Practice

This article includes an OpenSource program, ImageCaster, that shows how you can create a custom header. This program blasts out large files (like ISO CDROM images) to all receivers. It has several fields in the header: ID, size of the message, offset into the file, a checksum of the entire message, and a message digest of the data.

The program takes a file and breaks it up into parts and transmits it to a destination piece by piece. The destination can be either a peer (unicast) or a multicast address. The receiver subscribes, if necessary, to a multicast address and accepts each block of data. It disassembles the message and verifies data integrity. If all goes well, the receiver places the data in a file at the specified offset.

Datagrams and Strengthening Reliability

This is very useful to those, for example, who have tried to download an ISO CDROM image file (only 650MB!). You tune it to send a message (up to 64KB) every second. It may take longer on some systems, but the program almost ensures that you can get up to 64KB/s. Naturally, you can change the code to push it much faster.

Improving the Program

At present, the program only accepts messages from one server. The program does include the capability to accept inputs from several sources (multi-sourcing) all at different rates and different message sizes. Enabling multi-sourcing significantly reduces server load and maximizes your throughput.

This program is a little promiscuous: it accepts a message from any source. If you enable multi-sourcing, you may have a problem of degraded integrity, because it's difficult to verify the source of the message (spoofing/aliasing). The program also assumes that the file does not change while the sender transmits it.

One way to solve the spoofing (a falsified source address) and aliasing (an assumed source address) problems is to use encryption. The problem with sharing information is that not only does the receiver get the message, but the sniffer does too. This sniffer can doctor (or even completely garble) the data and retransmit it. The file you eventually get is less than useful. Using a public space like multicasting addresses compounds the problem.

Here is a little clever technique that you can use when sending to multiple designations: use public key encryption. Public key encryption (also known as asymmetric encryption) has two keys, one (public key) for encryption and the other (private key) for decryption. Normally, clients and servers use this process to establish a trusted connection (see the article on SSL).

Before the client and server establish the trusted connection they give each other their public keys. Using the public key, they encrypt a message that only the receiver can decrypt. This is very powerful and useful on the Internet, because it's very difficult to crack these encrypted messages, and *authenticates* (verifies the authenticity) the sender to the receiver.

In this multicasting program, you can get secure messaging by turning around the encryption process. Instead handing out the encryption key, hand out the decryption key. The multicasting server makes available the decryption key, because everyone should be able to read the data. But because nobody but the authenticated sender should generate the data, the server withholds the encryption key.

Using a reversed public key protocol, the receiving program gets the public key from the server and begins receiving the data. It decrypts the data using the key and stores the file as expected. Because encryption has its own built-in checksums, the program may no longer need the defined checksum and message digest. The only remaining limit is any network bottlenecks that can restrict the flow of data.

Not everyone has a T3 connection or even a 1MB/s connection, so setting the throughput too high may drowned many modem-connected receivers. The network, when hitting this data

Datagrams and Strengthening Reliability

logjam, may simply drop the incoming messages. This is worsened with a problem between multicasting and ports, which the first article described. When a program subscribes to a multicast address, the network does not search for only those packets that match address *and* port. Instead, all messages matching the address arrive to the receiving computer – regardless of the selected port value. Unfortunately, there currently is no solution for this problem. You as the code writer must be aware that your users may face this problem, and you may have to adjust the programs accordingly.

Solving Some Problems

All is not lost for those that have limited bandwidths. In fact, you can get the data you want with the supplied programs as long as the data throughput is lowered a little. Also, a new multicast technology called Small Group Multicast (SGM) is in the works and promises to solve many multicasting problems.

Datagrams and multicasting, when used in the right way, can reduce the load on our networks and increase the current throughput. Some applications that are currently written for TCP connection could fit easily in a reliable datagram protocol. If you want to know more about these techniques pick up *Linux Socket Programming* and visit the website, <http://www.cs.utah.edu/~swalton>.