Reliable Datagram Protocol

Gene Michael Stover

created Saturday, 2005 January 1
updated Thursday, 2005 January 13

Contents

1 What is this?

Since 1997, I've had an idea for a way to implement reliable datagrams efficiently & simply, in terms of a stream protocol such as TCP/IP. I'll discuss that idea here.

If I ever implement the idea, I'll do so in Lisp & C, & I'll discuss the source code here & provide links to it. (If I implement it, the source code will be licensed according to the Gnu Public License agreement or the Gnu Lesser Public License agreement.)

2 What is a reliable datagram?

A *datagram* is a chunk of data sent over a network & containing the information necessary to identify the destination. Contrast this with a chunk of data sent over a connection-oriented protocol, in which the destination is identified by the connection, not within the chunk of data.

A *reliable datagram* is a datagram that guarrantees it will be delivered exactly once except in the face of catastrophic network problems. Contrast this with most datagram protocols, which guarrantee that a given datagram will be delivered at most once.

There are some experimental protocols that offer reliable datagram services, but they are not widely used. I want to create a reliable datagram protocol at the user level so that it can be installed wherever required, maybe even self-contained within an application.

3 Why reliable datagrams?

Reliable datagrams are convenient for application programmers.

A datagram servic is more convenient than a stream service for many types of interprocess communication because it can place each message in a unit of data which can be treated atomically by the communicating programs. That unit of data is the datagram.

An unreliable datagram service, such as UDP, is inadequate as a messaging protocol because it requires the communicating programs to handle protocol errors themselves.

So many situations call for a reliable datagram service & an Application Programmer's Interface (API) for accessing the service conveniently.

4 Requirements

Here are the requirements I place on the reliable datagram service I'd like to implement.

1. Provides a reliable datagram service. Datagrams are delivered exactly once, or the sender & receiver are notified that there was a failure. It's okay if the notification is not delivered immediately after a datagram is sent.

2. Convenient API. Given the "address" of another process, sending a reliable datagram to it shouldn't be much more difficult than handing the data & the address to some kind of SEND-RELIABLE-DATAGRAM function.

3. Portable. It's built on POSIX networking functions such as `socket`, `bind`, `accept`, `connect`, `send`, `recv`, & `close`.

4. Does not require special permissions from the user or application. In other words, it's built on whatever networking services than an application can access. It does not require super-user priviledges so that it could, for example, implement its own protocol directly on IP or Ethernet. A standard user could install it, or an application run by that user could run its own semi-custom reliable datagram service; intervention from a super-user is unnecessary.

5. Reasonably low latency & high bandwidth. My reliable datagram protocol doesn't need to take esoteric steps to ensure efficiency, but it shouldn't do anything that's seriously inefficient.

6. Allows a process to be in communication with a very large number of other processes.

5 Application Programmer's Interface

5.1 What is a datagram?

Is a datagram a sequence of octets, with the destination or source address being closely related but not part of it, or is a datagram an object which contains a destination &/or source address & a sequence of octets?

A datagram that a program is processing is probably used with its sender's address much of the time. In Lisp, it would be simple enough to make a datagram be a sequence of octets & to associate it with an address in a list when necessary. C isn't as flexible; to associate the octets with the address, it will be most convenient to put them in a structure. I don't need to do that in Lisp, but I'd like to have similar APIs in both languages, so I'll make a datagram structure in both languages.

5.2 What is an address in reliable datagram space?

If a reliable datagram service multiplexes datagrams on a particular TCP port, a full address in reliable datagram space might be the host's Internet address, a TCP port number, & a reliable datagram port number.

Wait a minute. Back up. Here's an idea:

What if an address is a hostname & a queue name? Reliable datagrams could be implemented in terms of a service accessible from TCPMUX. The reliable datagram service could have a default TCPMUX name, but applications might have a custom reliable datagram service running, labeled with a custom name in TCPMUX.

When receiving datagrams, an application program gives the reliable datagram library the name of the queue(s) from which it wants to receive messages. Yes, multiple processes on a host could receive from the same queue if they all knew the name of the queue.

There would be no need to create a reliable datagram service object within the application program.

6 API For Lisp

I presume that all of these functions live in a package named CYBERTIGGYR-RELIA or something like that.

6.1 datagram

```
(defstruct datagram
  octets
  src-addy
  dst-addy)
```

A datagram contains the octets of a message, plus the address of the sender or receiver.

The octets are in a list, vector, or possibly some other sequence.

When sending a datagram, you must fill-in the destination address before calling SEND.

When receiving a datagram, the RECV function will fill-in the source address.

6.2 *status*

Other functions in this reliable datagram library bind error conditions to this global variable. I'll define those error values later. They will probably be symbols. If *STATUS* is bound to NIL, there has been no error since *STATUS* was last bound to NIL.

6.3 create-queue

`create-queue` *name* `=>` *name-or-nil*

Create an input queue. Until you create the queue, no one can send datagrams to it. There is no privacy for the receiver, though. Any process on this host may receive from the queue if they know the queue's name.

Name is a string. It identifies the queue. I need to define the characters that are permitted in the queue's name. It should probably allow the characters that may be in an URL. How about a maximum length?

Should the queue be private? If so, how to implement that?

Returns the normalized queue name (in case characters must be encoded) on success. If the queue can't be created, returns NIL. It is not an error if the queue already exists.

6.4 send

`send` *octets addy* `=>` good-or-bad

Sends the sequence of *octets* to the address *addy* (in reliable datagram space).

Octets may be a list of octets or a vector of octets. Maybe other types of sequences of octets are permissible.

Addy identifies a destination. It may be any of these:

- An URL of the form "relia:*//hostname/queue-name*", where *hostname* is an Internet hostname & *queue-name* is the name of a reliable datagram queue on the destination host.
- A two-element list whose FIRST is the hostname string of the destination host & whose SECOND is the queue name as a string.
- A two-element list whose FIRST is a list of the four parts of the Internet address of the destination & whose SECOND is the queue name. The four parts of the Internet address are in big-endian order.

Returns NIL on error (which might refer to a previously sent datagram). On success, returns true.

datagram is a DATAGRAM structure with its OCTETS & DST-ADDY members filled-in.

service is a reliable datagram service.

6.5 recv

`recv` *queue-name* `=>` *list-or-nil*

Attempts to receive the next available datagram from the service. If there is a datagram available in the queue, you get it as a list. Otherwise, you get NIL.

Queue-name is a string. It identifies the input queue on this host.

List-or-nil will be NIL if there are no datagrams ready for input. It'll also be NIL if there was an error. To distinguish between a no-datagrams situation & an error situation, examine *STATUS*.

If there is a datagram in the queue, you'll get a two-element list. The list's FIRST is a vector of the octets in the datagram. The list's SECOND identifies the sender. It will have one of the forms that may be used for the SEND function. (For the sanity of the application programmer, I should declare that it will have exactly one of those formats always, or there should be an optional argument which specifies which of the formats the SECOND should have.)

6.6 close-queue

`close-queue` *name*

7 API For C

I presume all these symbols exist in a module called RELIA or something like that. The way I program modules, that means these symbols in C code might be prefixed with "RELIA_".

7.1 errno

Holds the last error status or 0 if there hasn't been an error.

7.2 CreateQueue

```
char *
CreateQueue (name)
     char name[];
```

Creates a datagram input queue. Normalizes the queue's name, if necessary, & returns a dynamic copy of the normalized name. (Caller must release the name's memory with `xfree`.)

On error, returns `NULL`.

It is not an error if the queue already exists.

7.3 Send

```
int
Send (octets, length, addy, queue)
```

```
     char octets[/* length */];
     int length;
     struct in_addr *addy;
     char queue[];
```

Octets is a list of the octets in your datagram. *Length* is the number of octets in the list.

Addy is the Internet address of the destination hostname.

Queue is the name of the reliable datagram queue on the destination host.

Returns 0 if there have been no errors. Returns non-zero if this datagram or some predecessor couldn't be sent.

7.4 Recv

```
int
Recv (queue, octets, length, addy)
     char queue[];
     char octets[/* length */];
     int length;
     struct in_addr *addy;
```

If there is a datagram waiting in the queue, copies its octets into *octets*, but will not over-flow *octets*, assuming its length is *length*. Stuffs the sender's host address into **addy*. Returns the number of octets that were copied into *octets*.

If there is no datagram, returns a negative number but does not put an error into `errno` because this is not an error.

If there is an error, returns a negative number & puts an error code into `errno`.

7.5 CloseQueue

```
void
CloseQueue (name)
     char name[];
```

Testing one two three tea & jam.

8 Implementation Ideas

8.1 Per-Message TCP connection

One way to implement reliable datagrams is to create a TCP connection to the destination process, send the entire datagram, & close the TCP socket.

Advantages of this approach are:

- It's simple.

- Datagrams are very distinctly partitioned.
- It would allow a process to communicate with many other processes, ignoring per-process socket limits, because it uses only one socke at a time.

Disadvantages of this approach are:

- It's grossly inneficient because creating a TCP connection is expensive, & good TCP implementations restrict the bandwidth of a new connection to avoid the "Silly Window Syndrome" [1]. So unless the applications send very large datagrams, the through-put & latency they realize from the network would be nothing close to what it could theoretically deliver.
- Creating & destroying lots of TCP sockets can back-fire on systems which have long (or infinite) timeouts on TCP's FINWAIT2 state. This used to be a big problem, almost certain to happen, though since HTTP became popular, many network stacks have been improved so that this does not happen.

8.2 UDP with retransmits

Another way to implement a reliable datagram service is to build an error-detection & retransmission feature on terms of UDP. The reliable datagram service would handle the retransmits automatically; the applications which used the reliable datagram service would not need to worry about it.

Advantages of this technique include:

- It's simple & obvious.
- It satisfies the "communication with a very large number of other processes" requirement.

Disadvantages of this technique include:

- Simple error-detection logic is grossly inefficient. A naïve implementation would probably use acknowledgements on a per-datagram basis. This is stop-&-wait, which wastes most of the network's potential through-put. (The faster the network, the more through-put is wasted.) [4]
- More efficient error-detection logic, such as a windowing protocol [4], is non-trivial to implement, & windowing protocols are already conveniently available in the form of reliable connection-oriented protocols such as TCP.
- Firewalls will almost surely block UDP packets sent to all but a handfull of designated ports. (A reliable datagram protocol implemented in terms of TCP might still have a slim chance of making it through a firewall, but it'll be a better chance than the one implemented in terms of UDP.)

8.3 TCP connection per process

The reliable datagram service creates a single TCP connection per pair of processes that want to communicate. It multiplexes on that single TCP connection datagrams sent between the two processes. The reliable datagram protocol uses a single TCP connection for all messages between two processes. Messages passing in either direction go over the same TCP connection.

The reliable datagram service creates & destroyes the TCP connections internally, without worrying the application program.

If a given process communicates with multiple processes, the reliable datagram protocol will (probably) create a TCP connection for each of them. If the application program tries to send to a new process, & the reliable datagram service tries to create a new TCP connection but is alerted that it has reached its limit, it can silently close the least recently used connection & create the new one. This connection-swapping is hidden from the application program.

Advantages of this technique are:

- TCP provides reliability & efficiency.
- The TCP connections can be long-lived, so they can take full advantage of the network's through-put.
- Allows a process to communicate with a large number of other processes, but possibly not with a "very large" number of other processes.

Disadvantages of this technique are:

- I can't think of any.

A bunch of processes that want to communicate with each other (maybe they are the same application) could be programmed to use a particular TCP port on which to multiplex all their reliable datagrams.

8.4 TCP connection per host

This is like the "TCP connection per process (Section 8.3)" technique except that it creates a TCP connection between each pair of hosts that want to exchange reliable datagrams.

Just like the "TCP connection per process" technique, it multiplexes on a single TCP connection all the datagrams between a pair of processes, but unlike the other technique, this technique multiplexes the datagrams from all the processes on host A that are sent to or from host B.

Advantages of this technique include:

- Should have long-lived TCP connections, even longer than those of the "TCP connection per process (Section 8.3)" technique, thereby achieving slightly higher through-put.

- If the reliable datagram service were used by many processes (by being used by many applications), this technique would have even fewer TCP connections than would the "TCP connection per process (Section 8.3)", making it even less likely to bump into the per-process limit on TCP sockets, thereby allowing a process to communicate with a "very large" number of other processes.

Disadvantages of this technique include:

- If such a reliable datagram service were going to multiplex the datagrams from a bunch of unrelated processes, those processes might want the service to be run by a trusted user. That effectively, if not literally, conflicts with the "does not require special permissions" requirement I placed on the project.

  On the other hand, a bunch of related processes might start their own reliable datagram service on a TCP port that was compiled into them. They'd all use that instance of the service, & it could multiplex all the messages they sent or received. So it's only half a disadvantage, I'd say.

This is the technique I'll use if/when I implement a reliable datagram protocol.

9 C implementation

It's a library that can be linked into C programs.

The application program should not need to worry about the implementation of the reliable datagrams. In particular, it should not need to configure communication with the local reliable datagram service.

The library could communicate with the local reliable datagram service through unix datagram sockets when on unix or UDP on Windows or Macintosh.

10 Lisp implementation

It's a library that can be loaded into a Lisp program & used from it.

The application program will not need to do special configuration that relates to the implementation of reliable datagrams.

The library communicates with the local reliable datagram service through files because that is portable among Common Lisp.

A future version of the library could use UDP or TCP sockets on Lisp implementations that offer them, even if UDP & TCP are offered in non-portable ways.

11 Conclusion

I'll probably get around to implementing it some day.

A Glossary

A.1 TCP/IP

"TCP/IP" usually means the suite of network protocols that are built on the Internet Protocol. The two most notable players in that suite are TCP & UDP.

TCP is the Transmission Control Protocol. It is a reliable stream protocol. TCP is defined in RFC 793[2].

UDP is the User Datagram Protocol. It is an unreliable datagram protocol. UDP is defined in RFC 768[3].

B Other File Formats

- This document is available in multi-file HTML format at http://lisp-p.org/relia/.
- This document is available in Pointless Document Format (PDF) at http://lisp-p.org/relia/relia.pdf.

References

[1] Daniel C. Lynch Marshall T. Rose. *Internet System Handbook*. Addison-Wesley Publishing Company, Inc., 1993. ISBN 0-201-56741-5.

[2] J. Postel. Rfc 0739: Transmission control protocol. *available online*, September 1981. http://www.cis.ohio-state.edu/cs/Services/rfc/rfc-text/rfc0793.txt.

[3] J. Postel. Rfc 0768: User datagram protocol. *available online*, September 1981. http://www.cis.ohio-state.edu/cs/Services/rfc/rfc-text/rfc0768.txt.

[4] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 1989. ISBN 0-13-162959-X.