Current Status of RxTCP

Ken Hornstein
US Naval Research Laboratory

Current Project Goals

- Increase AFS performance to take advantages of modern hardware and networks, while tracking up-to-date code base (which includes things like 64-bit file sizes).
- First obvious target was tackling the transport protocol used by AFS (called Rx), a home-grown UDP based protocol developed in the late 80’s/early 90’s.
- Previous profiling and performance improvement attempts were hindered by poor Rx performance (hard to trace).
- Current direction is to implement new transport protocol over TCP, while keeping a similar API (some API changes are allowed, but a completely new API would involve too many changes across AFS).

Current Work To Date

Protocol has been designed by group of AFS developers, and has AFS community "buy in".

- At the Swedish AFS Hackathon, Jeff Hutzelman, Magnus Althorp, and myself worked on the RxTCP protocol design.
- Rx communication consists of connections (authentication abstraction) and multiple calls within each connection. Each call corresponds to a single RPC operation.
- Since connections are long-lived but calls are short-lived, it was decided to map a Rx connection to a TCP connection.
  - This prevents connections spending lots of time in slow-start, avoiding same problem with early HTTP protocol.

RxTCP Protocol Details

- All data is packetized and has a 12 byte header for each packet.
- Maximum packet length is variable (default is 8192 bytes, but can be changed by application).
- Packet header includes following fields:
  - Direction & "Last" bits (to indicate call end)
  - Packet type (data, new call, end call)
  - Total packet length
  - Call number (which may be zero if not associated with call).
- Including call number in header simplified the most common case (data packets).
- Original protocol design lacked some of these fields; header changed due to lessons learned during implementation.

Additional Protocol Details

- Multiple streams multiplexed over a single TCP connection can result in one writer unfairly consuming all available connection space.
- To alleviate this problem, RxTCP includes a flow control algorithm to balance bandwidth among consumers.
- Ideas taken from IETF BEEP protocol.
- Writers take round-robin turns writing packets.
- Data packets are acknowledged by receiver; as number of calls increase, window size for all calls gets smaller.

Additional Protocol Details

- Protocol includes concept of "reverse" connections.
  - Rx connection in opposite direction of TCP connection establishment.
  - Was suggested by people who have problems with NATs (fileserver needs to break callbacks to client).
  - Only partial support at this time; will likely require API extensions.
- Protocol provides traditional Rx connection identification to maintain API compatibility.
  - epoch - Start time that Rx library was initialized
  - cid - Start time of Rx connection
- Remove concept of Rx "channels" (limit of 4 outstanding calls per Rx connection).

Implementation Of RxTCP

- AFS uses two threading packages. One is the traditional pthreads interface, the other is called LWP and is a cooperative threading package which does it’s own stack pointer mangling.
- It was decided to focus the RxTCP effort exclusively on pthreads, since it was impossible to take advantage of multi-processor or multicore systems with LWP.
- Most fileservers deployed today use pthreads, as well as the volume server (most database servers are not, but they generally do not have large bandwidth requirements).
- Each operating system has it’s own type of in-kernel multiprocessing, but shares more features with pthreads than LWP, so seems like a sensible choice.

Implementing RxTCP Data Buffering

- The buffering scheme used by RxTCP went through a number of iterations in an attempt to minimize the number of data copies.
- A number of other packet-data-over-TCP protocols were examined (SSH & X-windows were two); all would read data into an intermediate buffer.
- Attempts to adopt a similar scheme and minimize the number of read() system calls were ultimately unsuccessful; keeping track of which call had which data in the buffer grew to be too complicated, and with multi-call support handling data in a ring buffer was nearly impossible.
- Considered implementing mbuf-like scheme, but realized Rx already had that and implementing that would probably result in significant performance penalty.

Final RxTCP Data Buffering Solution

- Upon connection creation, a buffer was created to hold RxTCP packet headers.

- When the connection was "empty" (no further data), _only_ the header was read into this buffer.

- If it was a data packet, the associated call was located, and the next read() system call reads data directly into the application buffer. If the application buffer is full, the data is buffered internally (and counted against the call’s window).

- As part of the same read() call (actually uses readv()), the header for the next packet is read into the packet header buffer. This saves system call overhead for the next packet.

RxTCP Pthread Interface

- Every time a new Rx connection (one TCP connection) is created, a dedicated worker thread is created to handle reading data from the connection file descriptor.
- This worker thread wakes up threads waiting for data in rx_Read() using standard pthread primitives. Write calls are protected against simultaneous calls, but could not come up with a scheme that increased efficiency (data is directly written from application buffer, using writev() to prepend packet header).
- This simple interface was chosen to get an implementation quickly. Should scale to reasonable number of connections, but connections numbering in the thousands will likely suffer from poor thread scheduling performance. May need to consider dispatching worker threads to deal with incoming packets.

Other Implementation Bits

- Most interfaces directly manipulated UDP packet queues; changed all of these interfaces to check for a TCP connection and make appropriate calls into the TCP module.
- Internals are sort of "hybrid"; while many internal fields were reused, a number were not used by RxTCP, so discovered via trial and error which fields were needed (if some fields were left uninitialized, Rx would crash).
- Original thinking was that TCP would always be tried first, then fall back to UDP. Later experience revised that expectation.

Testing RxTCP

- Original test suite (afsperf) was insufficient for needs.
  - Lacked easy way to add RxTCP items, and in beginning test program had to handle packets directly.
- Created new test suite (rxtest) to handle RxTCP exclusively.
- Test program can adjust window parameters, frame sizes, application write sizes, TCP buffer sizes, and total write length.

Test Results

- Original frame size of 4096 resulted in too many system calls; increasing to 8192 increased performance (frame sizes larger than that had no impact).
- Original window protocol per BEEP specifical resulted in a window size adjustment for every packet. This reduced performance by 20%!
- Switching to TCP model (window update every other packet) increased performance to near original levels (within a few percent).
- With the final buffering scheme and additional tuning, was able to achiece 90% of theoretical performance on Gig-E.
- Performance improvement over Rx showed that transport protocol design was sound.

Multi-Call Support

- Typical connection to fileserver has multiple calls simultanously over single Rx connection.
- Required debugging problems encountered with concurrency (had a data corruption problem on reads that took a long time to track down, related to multi-call handling).
- Additional support was added to rxtest to do simultaneous calls to exercise multi-call support.
- Tests with multi-call showed almost perfect linearity (transfer rate across all calls was within a few percentage points of a single-call transfer).

New Security Internal Interface

- The security library (rxkad) expected to be able to manipulate rx_packet structures directly.
- Since we did not use the rx_packet buffering interface, this made the existing interface difficult to use.
- After trying to shoehorn the existing security interface to deal with a stream protocol, it was finally decided to create a seperate set of security interface functions which could deal with a stream interface.
- This was implemented, and security is now equivalant to Rx.

New Rx APIs

- Has been ongoing desire to support IPv6 in Rx.
- Some commercial customers wanted way to bind to specific interfaces with Rx (currently the server binds to INADDR_ANY).
- We wanted the ability to specify UDP or TCP support.
- Had meeting in January to flesh out new Rx API functions.

Details of New API Functions

Existing API: rx_Init(u_short port)

- Only can take a port number, no specific interfaces or protocol.

New API: rx_InitAddrs(struct sockaddr_storage *, int *, int)

- Takes array of struct sockaddr_storage (different interfaces or protocols such as IPv6), array of protocol types (SOCK_DGRAM or SOCK_STREAM), and array size.

Existing API: rx_NewConnection(uint32, u_short port, ...)

- Only can take an IP address as connection address.

New API: rx_NewConnection(struct sockaddr_storage *, int *, int, ...)

- Same concept as other new API (array of addresses and protocols).

Current Work

- Integrating RxTCP implementation into AFS fileserver.
- Right now fileserver hangs on starting, which interacts poorly with the volserver.
- Implemented API extensions for RxTCP to solve these problems.
- Hope to resolve these problems soon.

Integration of RxTCP in AFS Client

- Likely to be the most challenging part, technically, since client is loaded into the OS kernel.

- In theory, should "just work" ... but I had the same theory about the AFS fileserver, and look how well that worked out.

- Only potential extra work at this time is to perform some housekeeping in the client to keep track of connection protocol to improve round-trip time during connection setup (e.g., don’t try TCP if the server only supports UDP).

- Given technical variables, will have estimate of time to complete after fileserver is done.

Any Questions?