Real-Time TCP for Embedded Devices

Aaron Gember
Marquette University, Dept. of Mathematics, Statistics, and Computer Science
1313 W. Wisconsin Ave., Milwaukee, WI 53233
agember@mscs.mu.edu
Advisor: Dr. Dennis Brylow

Problem & Motivation

An increase in both multimedia-based network services and embedded clients is presenting new challenges for existing network protocols. Time-sensitive services like Internet streaming media and Voice over IP require protocols which provide real-time performance. The protocols must also function well within the limited processing and memory constraints of embedded clients.

The goal of this work is to develop a real-time Transmission Control Protocol that satisfies the constraints of embedded devices while providing a transport mechanism for both general and time-sensitive data.

Transmission Control Protocol (TCP) [10] is the most widely used transport protocol [4]. It is a reliable network protocol appropriate for loss-sensitive streaming media. TCP's in-order delivery guarantees and network congestion control features are also sources of its merit. Many firewalls are configured to block User Datagram Protocol (UDP) packets, so TCP-based streaming is currently used for the majority of Internet streaming traffic [6]. Unfortunately, typical TCP implementations do not provide the timeliness necessary for streaming media. Compared to other network data, multimedia content is extremely sensitive to delay [5].

The processing and memory limitations of embedded systems create additional design constraints. First, the number of network stack layers needs to be minimized to avoid additional processing overhead. Second, the implementation should have a small code base and reuse the same code as much as possible. This can be achieved by implementing a commonly used transport protocol (in this case TCP) and extending it to also provide real- time communications. Lastly, embedded clients can only buffer small amounts of streaming data. The client needs to be able to continually receive data in a predictable time frame to provide suitable performance.

Background & Related Work

True real-time network performance requires the collaboration of the underlying network hardware. But control of this hardware is typically infeasible. The goal of real-time TCP and other real-time network protocols is to provide a best attempt at real-time communication using mechanisms at the transport layer of the network stack. Previous work on streaming media over TCP has focused on congestion control and its effect on delay. A model of the delays associated with TCP in relation to streaming media was created by [2] and [13]. TCP Friendly Rate Control (TFRC) is a protocol designed to reduce throughput variations caused by congestion [7].

Many streaming media applications use Real-time Transport Protocol (RTP)[11]. RTP is primarily used for multi-participant multimedia applications. The protocol is designed to be flexible and adaptable and "to provide the information required by a particular application" [11]. RTP provides application level framing and, like TCP, sequence numbering. Packet delivery is not guaranteed and RTP does not attempt retransmission of lost packets. The protocol typically sits on top of UDP, but TCP and Stream Control Transmission Protocol(SCTP) [12] have been used.

A number of application layer techniques have also been developed to address the challenges of streaming media. The technique known as *fast streaming* transmits media data faster than the encoding rate [6]. *Rate adaptation* changes between different encoding rates to compensate for network speed fluctuations [6].

The environment for this research is Embedded Xinu [3]: an educational embedded operating system running on Linksys WRT54GL wireless routers [9]. The operating system features thread scheduling, messaging passing, memory management, serial and TTY interfaces, a user shell, and a network stack. Both the WAN and LAN two network interfaces use direct memory access and have a ring buffer capable of holding 512 packets. Packets are demultiplexed and processed using multiple network receive threads. The network stack currently supports Address Resolution

Protocol (ARP), Internet Protocol (IP), Internet Control Message Protocol (ICMP), Dynamic Host Configuration Protocol (DHCP), UDP, and (as an outcome of this research) TCP. Overall, Embedded Xinu consists of a straightforward and succinct code base comprehensible by undergraduate students.

The regular TCP implementation in Xinu (the first phase of this research) consists of four categories of functions: receive, send, timer, and user-level. The implementation is approximately 2000 lines of C code. The main receive function is called by the IP layer of the network stack to process an incoming TCP packet; sub-functions demultiplex the packet to the appropriate TCP connection and process control flags and data based on the current connection state. Send functions prepare an outgoing TCP packet including control flags, acknowledgement and sequence numbers, window advertisement, options, and data. A send function in the IP layer is called to add additional headers and send the packet. Retransmission, zero-window persist, and 2-MSL timeout are handled by timer functions. A TCP timer thread triggers packet sending and connection state changes at scheduled intervals. Lastly, user-level functions provide a socket interface for opening, reading, writing, controlling, and closing TCP connections, paralleling the standard device paradigm used throughout Xinu.

UNIQUENESS OF THE APPROACH

Real-time TCP facilitates soft real-time constraints for networking with embedded devices. Only the receiver-side of a connection requires changes. Real-time TCP does not require changes to the sender-side. In the case of multimedia applications where data is typically only sent in one direction, only the multimedia client (the receiver-side) needs to change. This approach differs from other approaches, like [8], which require new protocols on both sides of the connection. Changing only one side of the connection limits the overhead associated with disseminating the new extensions.

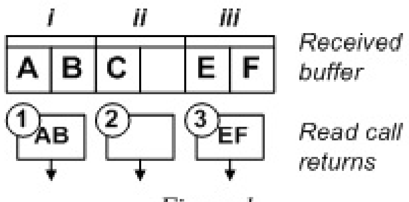

Figure 1

A best effort is made to receive data by a specific deadline. It is assumed that data received after a specific deadline is useless to the application; the missed data should be discarded. To accommodate the real-time constraints the data delivery guarantee of TCP is relaxed. When a deadlines occurs and data is unavailable, real-time TCP skips over the data, proceeding to receive data for the next deadline. No data is returned to the application. Figure 1 illustrates the concept. Real-time TCP begins to receive and acknowledge data the same as regular TCP, providing the first two octets of data to the application at deadline 1. When deadline 2 occurs, only one octet of data is available. Real-time TCP skips over both octets and returns no data to the application; regular TCP behavior resumes. By deadline 3 two more octets are received and provided to the application. The in-order delivery and retransmission features of regular TCP are maintained in real-time TCP.

Approximately 90 lines of additional C code and five additional control block fields are required for real-time TCP. Modifications are made to two functions: read and receive. The user-level read function is transformed to be non-blocking. The buffer length passed to read is interpreted as the length of a *frame* and only full frames are returned to the application. If a full frame of data is available in the buffer of received data, the read call removes the data from the buffer and returns the data as normal. When a full frame of data is not in the buffer of received data, the entire frame of data is skipped. An acknowledgement (ACK) is sent to the sender, acknowledging the full frame of data as if it was all received. No data is returned to the application. On the sender-side, the ACK is received and the sender assumes all data in the frame was received. Per regular TCP behavior, the sender no longer attempts to retransmit the data and moves forward to transmit the next octets of data following the frame.

Figure 2 illustrates the modified behavior. (The figure only shows one octet of data sent at a time for explanatory purposes; a received data buffer of one octet should also be assumed.) When a deadline occurs at time 3 and the buffer is empty, the receiver sends an ACK to indicate one octet was skipped; no data is returned to the application. Upon receiving this ACK, the sender transmits the next octet of data. The receiver buffers the data at time 5 and sends an ACK. The sender again transmits the next octet while the receiver simultaneously provides the buffered octet to the application. Notice that the *fake ACK* sent at time 3 to skip over the first octet of data causes the sender to send the next octet. If the fake ACK was not sent, the sender would have waited until the data was actually received and acknowledged at time 5, resulting in more missed deadlines.

The receive function is extended to handle a special case: the receiver skips over data which has not yet been sent. As discussed above, whenever the receiver-side skips over a frame it sends an ACK to the sender. However, it is possible the ACK acknowledges data which has not yet been sent. Since the sender is a normal TCP implementation it assumes the ACK must be a matter of confusion. It replies with an ACK which includes the sequence number of the next octet of data it plans to send; this sequence number will be less than the end of the frame which the receiver tried to skip over. The receive function on the receiver is modified to catch this "special" ACK and backtrack its attempt to skip over one or more frames of data. No improvement can be realized by real-time TCP when the sender-side has not sent a frame.

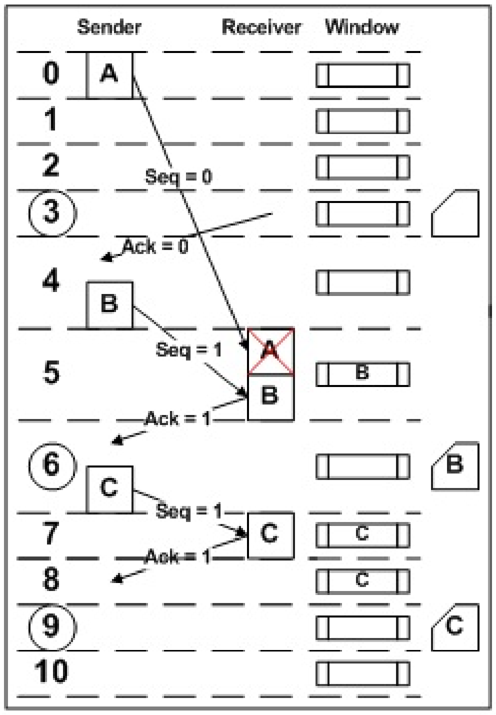

Figure 2

The modified receive behavior is illustrated in figure 3. The first octet of data is sent, acknowledged, received, and returned as normal for the deadline at time 3. When the deadline at time 6 occurs, the read function sends an ACK to skip over the frame (one octet of data). Since the sender has not yet sent the second octet, it replies at time 7 that the next octet it will send is the second octet. The receiver will backtrack its attempt to skip over the octet when the ACK is received at time 8. When the second octet arrives at time 9, just in time for the deadline, the second octet is returned to the application. If the second octet was not sent until much later, the receiver would have continually skipped over the second octet and then backtracked is attempt.

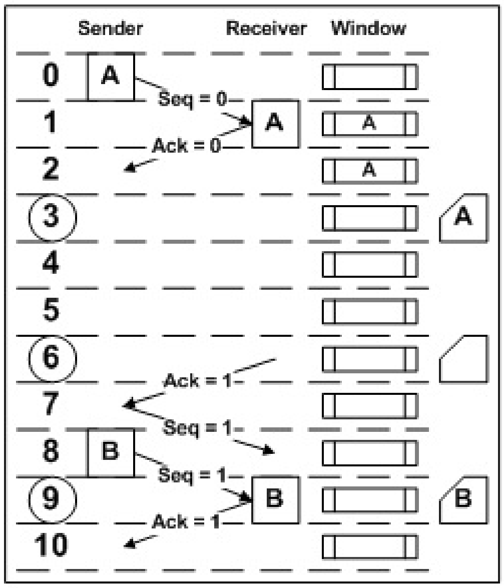

Figure 3

Results & Contributions

Benchmarking was performed using two routers running Embedded Xinu and a computer running the ns-2 [1] network simulator on FreeBSD. The *source router* contained a 719KB multimedia file and ran the regular TCP implementation in Embedded Xinu. The *destination router* ran the real-time TCP implementation and attempted to read 2KB of data every 250ms to match the encoding rate of the file. The destination router had a 4KB buffer for received data, a small size appropriate for an embedded client. Both routers had an initial zero-window persist timer of 3 seconds. Network delay and dropped packets between the source router and the destination router were emulated using ns-2. A 10ms delay existed between the two routers in both directions of communication. Emulation of 0%, 2%, 4%, 6%, 8%, and 10% dropped packets were used. Performance was measured based on the percentage of deadlines met over an average of three runs for each of the drop rates.

The benchmarking results are shown in figure 4. For the regular TCP implementation (not shown in the graph) only 1% of the deadlines were met for all packet drop rates. Regular TCP is focused on guaranteed data delivery, so it retransmits dropped data; by the time this data arrives the receiver has missed a deadline. A missed deadline occurs within the first few read calls because of the side- effects of a small buffer, described below. Once regular TCP misses the first deadline it falls behind and never catches up, resulting in only 1% of deadlines being met. The percentage of deadlines meet for the "skip over data" mechanism used by real-time TCP is much better than regular TCP. However, the performance is still much lower than expected. Only 63% to 67% of the deadlines are met. The expectation was to miss fewer deadlines than the percentage of packets dropped. The results show no correlation between percentage packet drop and percentage of deadlines met.

The poor performance can be partially explained by the concept of the TCP sliding window. The TCP sliding window is a range of octets the receiver (or destination router) is willing to accept from the sender (or source router). Window size is determined by the receiver's buffer size and congestion control algorithms. Each ACK includes the current size of the receiver's window. When the receiver advertises a window size of zero, the TCP specification dictates the sender should enter a *zero-window persist* state. In this state, the sender is responsible for occasionally sending an ACK to probe the receiver and determine if the window size has become non-zero. When the advertised window size becomes non-zero, the sender can leave the zero-window persist state and resume sending data to the receiver.

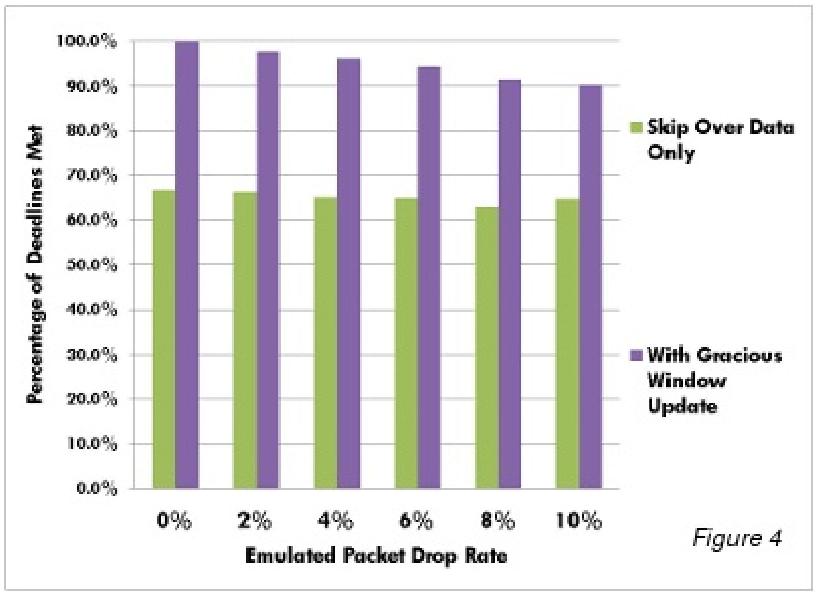

Figure 4

The destination router running real-time TCP is configured to have a small buffer - only 4 KB. This buffer is rapidly filled and the window size becomes zero. The last ACK the destination router sends contains a window advertisement of zero, so the source router enters the zero-window persist state, scheduling a timer to probe the receiver 3 seconds later. In the meantime, the destination router consumes two frames of data (totaling 4KB). When it tries to consume the third frame of data, at time 750ms, the buffer is empty. The destination router skips over data, sending an ACK which includes a window advertisement for a window size of 4KB since the buffer is now completely empty and available for more data. The source router leaves the zero-window persist state before its timer is triggered and again sends data to refill the buffer on the destination router. Thus a deadline is missed approximately every third frame, accounting for the results shown in figure 4.

To help remedy the issue, an optional feature known as gracious window update can be included in the real-time TCP implementation. The receiver sends an ACK, including a window update, to the sender every time a read call on the receiver-side results in the window becoming non-zero. Gracious window is not required by the TCP specification, although FreeBSD and Linux both implement this feature. Figure 4 shows the improvements in percentage of deadlines met for real-time TCP's skip over data mechanism "with gracious window update." It is important to note that adding only gracious window update to regular TCP will not yield much performance improvement. As soon as an ACK with a gracious window update is dropped, TCP will begin to fall behind and all further deadlines will be missed.

One concern with gracious window update in real-time TCP is its effect on congestion. Congestion control is one of TCP's merits which real-time TCP should preserve. Sending an ACK every time the buffer becomes non-zero will result in many additional ACKs. If the network is congested, this will exacerbate the situation. Future work will focus on developing an algorithm to determine when a gracious window update is required. Round-trip time and an understanding of when the next deadline will occur are suspected to be key pieces of information for developing this algorithm.

Additional benchmarking is also necessary as future work. The effect of network delay on real-time TCP should be measured and changes made if necessary. Increases and decreases in buffer size are also an important factor to consider. Lastly, real-time TCP should be benchmarked against other protocols like UDP and RTP.

Real-time TCP for embedded devices meets the needs of both real-time and general network traffic while satisfying the constraints of embedded systems. The minimal additional code overhead makes real-time TCP an ideal candidate for a system that also requires regular TCP. In keeping with the straightforward design of Embedded Xinu, real-time TCP is design with simplicity in mind. As a positive result of lower than expected performance, real-time TCP has illustrated the importance of gracious window update. Also, it has exposed network emulation as a potential new research direction for Embedded Xinu. Real-time TCP has room for improvement, but its unique approach makes it a well-suited transport protocol for real-time networking needs.

References

[1] The network simulator - ns-2. http://www.isi.edu/nsnam/ns.
[2] E. Brosh, S. A. Baset, D. Rubenstein, and H. Schulzrinne. The delay- friendliness of TCP. In SIGMETRICS '08: Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pages 49-60, New York, NY, USA, 2008. ACM.
[3] D. Brylow. An experimental laboratory environment for teaching embedded operating systems. SIGCSE Bulletin, 40(1):192-196, 2008.
[4] A. Dunkels. Full TCP/IP for 8-bit architectures. In MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services, pages 85-98, New York, NY, USA, 2003. ACM.
[5] P. Fouliras. On RTP filtering for network traffic reduction. In MoMM '08: Proceedings of the 6 th International Conference on Advances in Mobile Computing and Multimedia, pages 356-359, New York, NY, USA, 2008. ACM.
[6] L. Guo, E. Tan, S. Chen, Z. Xiao, O. Spatscheck, and X. Zhang. Delving into internet streaming media delivery: a quality and resource utilization perspective. In IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement, pages 217-230, New York, NY, USA, 2006. ACM.
[7] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP friendly rate control (TFRC). RFC 3448, Jan 2003.
[8] S. Liang and D. Cheriton. TCP-RTM: Using TCP for real time multimedia applications. In International Conference on Network Protocols, 2002.
[9] Linksys. WRT54GL wireless-G broadband router. http://www.linksys.com.
[10] J. Postel. Transmission control protocol. RFC 793, Sep 1981.
[11] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. RFC 3550, Jul 2003.
[12] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream control transmission protocol. RFC 2960, Oct 2000.
[13] B. Wang, J. Kurose, P. Shenoy, and D. Towsley. Multimedia streaming via TCP: An analytic performance study. ACM Trans. Multimedia Comput. Commun. Appl., 4(2):1-22, 2008.