Implementing SCTP to support WebRTC data channels in pure Rust

David Simmons
Boulder/Denver Rust Meetup
September 18, 2018
About me

- Freelance software engineer
- Worked for many years in the consumer electronics industry
- C, C++, Linux device drivers, audio/video
- Long-time systems programmer interested in network protocols

@simmons
simmons
https://davidsimmons.com/
Outline

- Goals
- The problem domain
  - Peer-to-peer networking and WebRTC
  - SCTP: Stream Control Transport Protocol
- Implementing SCTP in Rust
  - Approach
  - Design
  - Implementation
  - Testing
  - The Future
- Demo time!
Goals

• Long-term: A set of crates that provides a complete solution for using WebRTC outside of a web browser, thus providing a peer-to-peer networking stack for devices and apps.

• Short-term: Support WebRTC data channels with a Rust implementation of the Stream Control Transport Protocol (SCTP).

• The stack should be pure Rust, with the exception of cryptography.
Outline

• Goals
• The problem domain
  • Peer-to-peer networking and WebRTC
  • SCTP: Stream Control Transport Protocol
• Implementing SCTP in Rust
  • Approach
  • Design
  • Implementation
  • Testing
  • The Future
• Demo time!
Why peer-to-peer networking?

- Technical reasons
  - Lower latency
  - Reduce bandwidth costs
- Political reasons
  - Theoretically possible to build overlay networks for privacy, anonymity and “routing around censorship” (e.g. I2P)
  - Self-hosting to reduce dependency on third parties
- Creative reasons
  - Experimenting with novel decentralized systems (e.g. distributed hash tables (DHTs), cryptocurrencies)
  - Enable the development of network applications that have yet to be imagined
Peer-to-peer applications

Conferencing

Multiplayer games

File transfers

Security cameras
In an ideal world, we could connect to peers as simply as we connect to servers.
- Consumer broadband routers use Network Address Translation (NAT) to provide private address spaces, translate addresses, and protect the network with simple stateful firewalls.
- Direct communication between hosts is problematic when each is behind a NAT.
The UDP hole punching technique works around NATs by each host independently initiating communication, thus establishing the required NAT mappings on both routers.

Unfortunately, this means re-inventing the world on top of UDP.

"TCP simultaneous open" is a thing, but is unreliable in practice.

* - It's actually more complicated than this.
IETF standardization of NAT Traversal

- Bespoke NAT traversal systems were invented and re-invented for years (e.g. Skype, circa 2003).
- In recent years, the Internet Engineering Task Force (IETF) has developed standards for NAT traversal.
  - STUN (RFC 3489, 2003) - Discover one's public IP and NAT situation.
  - TURN (RFC 5766, 2010) - Relay traffic as a fallback.
  - ICE (RFC 5245, 2010) - A system for negotiating address candidates and NAT-traversed UDP flow.
- NAT traversal is a kludge, but at least now it’s a standardized kludge.
Enter WebRTC

• From my perspective as a systems programmer, WebRTC bundles all the best-known methods for peer-to-peer networking, and provides a common target for interoperability.

• Invented at Google in 2011, standardized by the W3C and IETF (RFC 7478 and others, 2015-)

• Like many things that start with the word “Web”, WebRTC has ambitions beyond web browsers.
Re-inventing TCP

- We need to re-invent everything we lost when we ditched TCP in favor of UDP hole punching.
  - Reliability (when desired)
  - Ordered delivery (when desired)
  - Congestion control
- One solution would be TCP-over-UDP.
- WebRTC instead picked SCTP as a “better TCP”.
Specification graph to support data channels

RFC 4960: Stream Control Transmission Protocol
RFC 3758: SCTP Partial Reliability Extension
RFC 7496: Additional Policies for the Partially Reliable SCTP Extension
RFC 6525: SCTP Stream Reconfiguration
RFC 5061: Dynamic Address Reconfiguration
RFC 4820: Padding Chunk and Parameter for SCTP
RFC 8260: Stream Schedulers and User Message Interleaving for SCTP
RFC 8445: Interactive Connectivity Establishment (ICE)
RFC 5389: Session Traversal Utilities for NAT (STUN)
RFC 5766: Traversal Using Relays around NAT (TURN)
RFC 4566: SDP: Session Description Protocol
RFC 3264: An Offer/Answer Model with SDP
draft-ietf-rtcweb-data-channel-13: WebRTC Data Channels
draft-ietf-rtcweb-data-protocol-09: WebRTC Data Channel Establishment Protocol
draft-ietf-mmusic-ice-sip-sdp-21: Session Description Protocol (SDP) Offer/Answer procedures for ICE
draft-ietf-rtcweb-jsep-24: JavaScript Session Establishment Protocol
draft-ietf-mmusic-sctp-sdp-26: SDP Offer/Answer Procedures For SCTP over DTLS Transport
draft-ietf-rtcweb-transports-17: Transports for WebRTC
draft-ietf-ice-trickle-21: Trickle ICE: Incremental Provisioning of Candidates for the ICE Protocol

845 pages!!!!1! 😱😧😦😱😢😩
WebRTC protocol stack

Communicating with a peer is as simple as this.
Communicating with a peer is as simple as this.
Outline

- Goals
- The problem domain
  - Peer-to-peer networking and WebRTC
  - **SCTP: Stream Control Transport Protocol**
- Implementing SCTP in Rust
  - Approach
  - Design
  - Implementation
  - Testing
  - The Future
- Demo time!
What is SCTP?

- A transport-layer protocol that in theory is layered on top of IP, same as UDP or TCP.
- Invented in 2000 by the telecommunications industry.
  - When you make a phone call on your LTE phone, it's using SCTP to establish the call.
- More flexible than TCP: Configurable reliability, configurable ordered/unordered delivery, multiplexed streams, etc.
- Due to protocol ossification and poor support in OS and routers, SCTP has never been practical to use on the Internet as originally intended.
- WebRTC bypasses protocol ossification and OS support issues by encapsulating SCTP in UDP. (Actually, SCTP-over-DTLS-over-ICE-over-UDP.)
Why re-implement SCTP?

- Education
  - Experience with transport protocols
  - Experience with Rust networking (tokio, futures)
- Most everyone re-uses the same C-based SCTP implementation from FreeBSD (libusrsctp), and it's good to have options.
- A pure Rust implementation might (some day) prove more reliable than the C implementation.
Why not QUIC?

• It’s possible that QUIC may indeed eventually replace SCTP for WebRTC data needs.

• In its current form, QUIC may not provide the feature set needed for WebRTC. It may not be suitable for non-reliable or partially reliable data channels.

• There does appear to be work in progress on both the QUIC and WebRTC fronts to remove any obstacles to making QUIC a first-class citizen of the WebRTC world.

• Today, in 2018, you need SCTP for WebRTC. Current proposals for using QUIC in a WebRTC context seem to provide it as a separate bolt-on API, and not a direct replacement for the existing Data Channel feature.
Outline

• Goals
• The problem domain
  • Peer-to-peer networking and WebRTC
  • SCTP: Stream Control Transport Protocol
• Implementing SCTP in Rust
  • Approach
    • Design
    • Implementation
    • Testing
  • The Future
• Demo time!
Set realistic goals

- Limit implementation to only the feature set needed by WebRTC.
  - Don’t support multi-homing
  - Do support the SCTP extensions required by WebRTC
- Stick to stable/published tools and crates
- In the early stages, prefer boring code over cleverness
  - Use Box, .clone(), etc. for now and optimize later.
  - Simple, straightforward use of traits and lifetimes.
- Road map:
  1. Produce a minimal proof-of-concept to gain confidence
  2. Produce a correct implementation
  3. Optimize
Rusty tools

• Use nom to parse packets.
  • Is this really the most efficient way?
  • It would be nice to have a tool that would both parse and synthesize based on a format description.
• Tokio
Rapidly evolving crates

- **tokio-core → tokio**
  - Still using the legacy/deprecated tokio-core API
  - Tokio now defaults to a multi-threaded reactor, so futures must be thread-friendly (Rc → Arc, etc.)

- **futures 0.1 → ?**
  - Using the latest published futures crate — version 0.1
  - futures 0.2.x was briefly published, then yanked, to much drama.
  - When stable and published, migrate to 0.2, 0.3, async/await.
  - futures::sync::mpsc race condition in 0.1

- **tokio-timer 0.1 → 0.2**
  - 0.1 has 100ms clock granularity; 0.2 has 1ms granularity
Outline

• Goals
• The problem domain
  • Peer-to-peer networking and WebRTC
  • SCTP: Stream Control Transport Protocol
• Implementing SCTP in Rust
  • Approach
  • Design
    • Implementation
    • Testing
    • The Future
• Demo time!
Stack design

- API
  - Asynchronous interface via Tokio
  - Synchronous API provided on top of the async API
- Configurable lower-layer protocol (LLP)
  - SCTP-over-UDP for testing with libusrsctp utilities
  - SCTP-over-DTLS-over-UDP for WebRTC
Futures design

• Currently, a single task is used for the SCTP stack and its association futures.
Outline

• Goals
• The problem domain
  • Peer-to-peer networking and WebRTC
  • SCTP: Stream Control Transport Protocol
• Implementing SCTP in Rust
  • Approach
  • Design
  • Implementation
• Testing
• The Future
• Demo time!
Associations

• Composing an association out of smaller futures or components is difficult due to the large and interconnected state.

• It’s hard to break out cross-cutting concerns like retransmission and congestion control because they touch so many other parts of association state. (But this deserves more thinking!)

• Examples of association state:
  - SCTP association state (Established, CookieEcho, etc.)
  - Network 4-tuple (src/dst addresses/ports)
  - Stream counts
  - Queues: send, sent, receive, reassembly
  - Local and peer verification tags
  - Sending TSN, receiving TSN high water mark
  - Current calculated peer receiver window
  - Retransmission measurements
  - Numerous timers (timeouts, rtx, ...)
  - Many other things!
Message reassembly

- Multiplexing and optional ordering make reassembly a bit more complex than in TCP.
Memory efficiency

- The total payload size of all receive buffers (regardless of where they are enqueued) is tracked to enforce window size.
- Some buffers are reference counted for efficiency, but need to switch to the bytes crate.
- Memory for incoming packets is simply allocated on the heap.
- It turns out that even Linux heap-allocates for incoming data:

```c
/* We do our best to align skb_shared_info on a separate cache line. It usually works because kmalloc(X > SMP_CACHE_BYTES) gives aligned memory blocks, unless SLUB/SLAB debug is enabled. */
size = SKB_DATAALIGN(size);
size += SKB_DATAALIGN(sizeof(struct skb_shared_info));
data = kmalloc_reserve(size, GFP_MASK, node, &pfmemalloc);
```

— v4.12 net/core/skbuff.c line 231
Outline

- Goals
- The problem domain
  - Peer-to-peer networking and WebRTC
  - SCTP: Stream Control Transport Protocol
- Implementing SCTP in Rust
  - Approach
  - Design
  - Implementation
- **Testing**
  - The Future
- Demo time!
Unit tests

- There are lots of unit tests to cover the basics:
  - Parsing and synthesizing parameters, chunks, error causes, and packets.
  - Serial number arithmetic
  - Reassembly queues
  - Buffer management
- When testing with random values, seeded RNGs are used for reproducibility.
- More complex aspects such as the SCTP state machine cannot be easily tested in unit tests.
Simulation testing (1/2)

- A simulation framework supports integration tests where multiple stack instances communicate with each other.
- The downside to simulation tests is they only prove that the stack can interoperate with itself, not that it correctly implements SCTP.
- Simulations work by providing each stack with a custom lower-layer protocol that knows how to route packets in-memory:
Simulation testing (2/2)

- A pause/resume allows testing time-sensitive scenarios such as simultaneous shutdown.

- The arrangement of futures in the simulation looks like this:
Outline

• Goals
• The problem domain
  • Peer-to-peer networking and WebRTC
  • SCTP: Stream Control Transport Protocol
• Implementing SCTP in Rust
  • Approach
  • Design
  • Implementation
  • Testing
• The Future
• Demo time!
Risks

- Congestion control
  - If this isn’t correct, it can cause grief for other network users.
- Bugs in transport protocols can leave systems vulnerable to denial-of-service attacks.

Linux kernel bug: TCP flaw lets remote attackers stall devices with tiny DoS attack

'SegmentSmack' Linux bug gives a remote attacker the means to knock out a system with minimal traffic.
Performance concerns

- Using UDP means performing a system call (and incurring the associated context switch costs) for every single datagram received and sent.

- Meltdown mitigations are likely to amplify this cost.

- Linux provides `recvmsg()` and `sendmsg()` system calls for operating on more than one datagram at a time. Maybe Mio could leverage this on Linux and any other operating systems that provide a similar API?
Debugging

- The complex nature of transport protocols allow for a great many bugs that the Rust compiler can't save us from, so testing and debugging will likely be a major effort even after functional completion.
Demo time!