SCTP as a First-Class Transport in Go on Linux: An Implementation and Interoperability Study

Olivier Van Acker
Independent Researcher
London, United Kingdom
olivier@robotmotel.com

***Abstract*—This paper revisits a 2013 SCTP-in-Go implementation and re-establishes the design on Linux in a modern Go runtime tree. The central claim is that SCTP should be treated as a first-class transport protocol next to TCP and UDP, not as an exotic add-on. The argument is architectural and empirical: Internet protocol layering was designed to evolve, SCTP is a mature IETF transport, and practical integration in a contemporary runtime is straightforward when the networking stack is already protocol-agnostic at key extension points. We describe the Linux integration in Go's `net` package, show API parity with existing TCP/UDP patterns, and evaluate interoperability against an independent C++ endpoint built on Linux SCTP APIs. Results show that the implementation passes in-tree SCTP tests and bilateral Go¡-¿C++ interop scenarios while preserving SCTP-specific metadata such as stream identifier and PPID. The outcome supports a broader position: modern language runtimes should expose SCTP as a standard transport option for message-oriented and resilience-aware applications rather than forcing all designs through TCP or UDP.**

***Index Terms*—SCTP, Go runtime, Linux networking, transport protocols, interoperability, sockets API**

I. Introduction

Transport protocol choice in many software stacks has narrowed to TCP or UDP, even though Internet architecture never required such a binary choice. The original internetworking design emphasized extensibility and evolution at protocol boundaries [1]–[3]. SCTP was standardized to provide reliable, congestion-controlled, message-oriented transport with features that are difficult or expensive to emulate correctly above TCP or UDP [4]–[6].

This paper revisits an earlier 2013 implementation effort and updates the work for Linux and a modern Go runtime source tree. The goal is not to present SCTP as "new", but to demonstrate that it can be integrated into a mainstream language runtime with the same first-class treatment given to TCP and UDP. In this context, first-class means:

- explicit address and connection types in the standard networking package,
- normal resolver, dial, and listen flows,
- direct support for protocol-specific metadata and controls,
- test coverage and interoperability validation against an independent implementation.

The implementation in this repository adds SCTP support directly inside Go's `net` package on Linux, including public API surface (`SCTPAddr`, `SCTPConn`, `DialSCTP`, `ListenSCTP`), Linux data-path support for `sendmsg`/`recvmsg`, and targeted test and interop harnesses.

The paper's central thesis is that SCTP belongs next to TCP and UDP in modern runtime APIs because:

1) Internet protocol architecture is intentionally extensible [2], [3].
2) SCTP is a mature standards-track transport with broad, production-relevant semantics [4], [5], [7], [8].
3) Runtime integration is technically modest when implemented through existing socket abstraction seams.
4) Real interoperability with C++ Linux SCTP endpoints demonstrates practical viability.

The remainder of this paper builds this argument from standards context, implementation design, and measured evaluation.

II. Why SCTP Should Be First-Class

A. Architectural Basis: IP Was Designed to Evolve

The Internet stack has long separated network and transport concerns to permit independent evolution [1], [2], [9]. RFC 1958 makes this explicit: successful protocol design assumes change and adaptation over time [3]. Treating TCP and UDP as immutable endpoints of transport evolution is therefore an operational habit, not an architectural law.

B. SCTP Is Not Experimental

SCTP is a standards-track transport protocol with a mature socket API and long-running implementations [4], [5], [10]. Relative to canonical TCP and UDP definitions [11], [12], it combines properties that many applications need:

- reliable, congestion-aware transport like TCP,
- preservation of message boundaries unlike TCP,
- multistreaming to reduce head-of-line coupling,
- optional multihoming and failover support at transport layer,
- protocol-level extensibility (e.g., partial reliability, schedulers).

These are not theoretical features; they are codified in the base protocol and extensions [7], [8], [13], [14].

C. Industry Relevance

SCTP is also used as the transport substrate for WebRTC data channels [15], [16]. This matters for two reasons: first, it disproves the claim that SCTP has no mainstream deployment; second, it shows that modern systems already depend on SCTP semantics when message orientation and stream multiplexing are required.

D. Comparison with TCP and UDP

Table I summarizes why SCTP deserves parity in language runtime APIs.

TABLE I
TRANSPORT FEATURE COMPARISON

Feature	TCP	UDP	SCTP
Reliable delivery	Yes	No	Yes
Message boundaries	No	Yes	Yes
Built-in multistreaming	No	No	Yes
Transport multihoming	No	No	Yes
Congestion control	Yes	No	Yes
Socket API standardization	Yes	Yes	Yes

The practical conclusion is straightforward: if TCP and UDP are exposed natively in a runtime's core networking API, SCTP can be justified by the same criterion used for those protocols, namely broad utility plus standards-based interoperability.

III. DESIGN AND INTEGRATION IN THE GO RUNTIME

A. Integration Goal

The implementation target is parity with existing `net` package transport patterns: address resolution, dial/listen entry points, and connection methods should look and behave like TCP/UDP equivalents while exposing SCTP-specific controls where required.

B. Public API Additions

The Linux implementation introduces SCTP-specific types and functions in `src/net/sctpsock.go`, including:

- `type SCTPAddr { IP net.IP; Port int; Zone string }`
- `ResolveSCTPAddr(network, address)`
- `DialSCTP(network, laddr, raddr)`
- `ListenSCTP(network, laddr)`
- `type SCTPConn` with `ReadFromSCTP`, `WriteToSCTP`, `SetNoDelay`, `SetInitOptions`, `SubscribeEvents`

These additions mirror established Go networking style and maintain compatibility with common `Conn`/`PacketConn` usage.

C. Runtime Touchpoints

Implementation is localized to existing extensibility seams in `net`:

- Resolver and network parsing integration in `src/net/ipsock.go`
- Dial/listen dispatch integration in `src/net/dial.go`
- Linux SCTP socket operations in `src/net/sctpsock_posix.go`
- Linux SCTP constants and control-message marshaling in `src/net/sctpsock_linux.go`
- Stubbed behavior for unsupported targets in `src/net/sctpsock_stub.go`

This is the core evidence for integration ease: SCTP support was added by extending existing abstractions rather than redesigning the runtime networking architecture.

D. Data Path and Metadata

The implementation uses `SOCK_SEQPACKET` with `IPPROTO_SCTP`. Message transmission and reception are built on `sendmsg`/`recvmsg`, with ancillary control messages carrying SCTP metadata (`SCTP_SNDINFO`, `SCTP_RCVINFO`). This preserves message orientation while exposing stream and PPID metadata to applications.

E. One-to-Many First

The current design emphasizes one-to-many SCTP semantics for Linux. `DialSCTP` uses an unconnected socket model with remote destination retained for default sends, and `ListenSCTP` follows passive receive behavior on `SOCK_SEQPACKET`. This keeps the API simple while supporting key SCTP behavior.

Integration map in Go `net` package
1) Parse network: `sctp`, `sctp4`, `sctp6`
2) Resolve endpoint: `SCTPAddr`
3) Dial/listen dispatch in shared socket path
4) Linux SCTP options and cmsg handlers
5) Public `SCTPConn` methods for metadata-aware I/O

Fig. 1. SCTP follows existing TCP/UDP extension seams in Go networking internals.

IV. EVALUATION METHOD

A. Objectives

The evaluation verifies two claims:

1) SCTP is integrated as a first-class runtime transport, not as an out-of-tree shim.
2) The implementation interoperates with an independent C++ SCTP endpoint on Linux.

B. Environment

Measurements in this paper were captured on:

- OS: Ubuntu 25.10, Linux kernel 6.17.0-12-generic
- Architecture: `linux/amd64`
- Go tree: in-repo build, version `go1.27-devel_c9cbeb0`
- SCTP kernel module enabled (`sctp`)

C. Go Package Tests

Targeted tests in `net` were executed:

- `TestSCTPLoopbackReadWrite`
- `TestSCTPUnsupportedOnBadNetwork`
- `TestResolveSCTPAddrUnknownNetwork`
- `TestParseNetworkSCTP`
- `TestSCTPAddrString`

These tests validate local SCTP I/O behavior, error semantics, and network/address parsing paths.

D. Go¡-¿C++ Interoperability Matrix

Interop uses the repository harness at `misc/sctp-interop/harness/run_matrix.sh` and covers:

1) Go server receiving from C++ client.
2) C++ server receiving from Go client.

Both scenarios verify payload integrity and SCTP metadata transport (stream ID and PPID). The C++ side uses Linux SCTP userspace API calls and `recvmsg` ancillary parsing.

E. Timing Measurement

The complete matrix runner was executed 20 times. Wall-clock duration per run was recorded from shell timestamps. This provides a lightweight integration performance indicator for repeated bring-up, cross-language exchange, and teardown under the same configuration.

V. Results

A. Go In-Tree SCTP Tests

All targeted Go `net` SCTP tests passed in a single run. The test run reported successful socket creation and loopback communication on `AF_INET/SOCK_SEQPACKET/IPPROTO_SCTP`.

B. Interop Matrix Outcome

Both interoperability directions passed:

- Go server received C++ payload with expected stream and PPID.
- C++ server received Go payload with expected stream and PPID.

Table II summarizes the observed data-plane correctness.

TABLE II
OBSERVED GO¡-¿C++ SCTP INTEROPERABILITY

Scenario	Observed output
Go server ¡- C++ client	`stream=3 ppid=101 payload=cpp-to-go`
C++ server ¡- Go client	`stream=4 ppid=202 payload=go-to-cpp`

C. Repeated Matrix Runtime

For 20 repeated runs of the full interop matrix:

- mean runtime: 2.187 037 s
- minimum runtime: 2.177 443 s
- maximum runtime: 2.203 883 s
- standard deviation: 0.006 289 s

The low variation suggests stable harness behavior and reproducible cross-language SCTP exchange in the tested environment.

20-run matrix timing summary
Mean: 2.187 s
Min/Max: 2.177 s / 2.204 s
Std. dev.: 0.006 s
Data source: `paper/repro/data/interop_matrix_runtime_2`

Fig. 2. Runtime stability of the Go¡-¿C++ SCTP matrix on Linux.

D. Interpretation

The results support the implementation claim:

1) SCTP behaves as an ordinary transport option inside Go's standard networking paths.
2) SCTP-specific metadata survives end-to-end across independent implementations.
3) Integration and validation can be automated with the same workflow style used for other runtime networking features.

VI. Related Work

The original SCTP motivation and protocol model are well documented in standards and reference texts [4], [6]. The socket API formalization in RFC 6458 is directly relevant to runtime and language binding design [5]. Extensions such as partial reliability and stream scheduling further differentiate SCTP from TCP-centric abstractions [7], [8].

Several prior studies evaluated SCTP capabilities. Iyengar et al. explored concurrent multipath transfer over SCTP multihoming [17]. Penoff et al. described a portable userspace SCTP stack and portability/performance trade-offs [18]. These works support the view that SCTP is both implementable and practically useful across deployment contexts.

More generally, transport evolution research has shown how deployment friction can ossify protocol diversity [19]. This paper contributes from a runtime-engineering perspective: language-level first-class APIs materially reduce that friction by making non-TCP transports available through familiar interfaces.

Finally, WebRTC data channels demonstrate broad production exposure of SCTP semantics in modern applications [15], [16]. This real-world adoption weakens arguments that SCTP is niche or obsolete.

VII. Discussion

A. Why This Supports "First-Class" Status

The implementation and evaluation provide concrete evidence for first-class SCTP treatment in a modern runtime:

- **API symmetry:** SCTP follows the same naming and usage patterns as TCP/UDP in Go's `net` package.
- **Internal symmetry:** SCTP plugs into existing resolver, dial, listen, and file-descriptor workflows.
- **Interoperability:** wire-level behavior is compatible with independent Linux C++ SCTP endpoints.
- **Operational symmetry:** tests and harness execution integrate into standard CI-style workflows.

This is precisely what "first-class" should mean in a runtime context.

B. The Extensibility Argument

Internet architecture intentionally supports protocol evolution [1]–[3]. Treating SCTP as a default-capable option is consistent with this model and avoids unnecessary application-layer reimplementation of transport semantics. This is aligned with end-to-end design reasoning: semantics that belong at transport should not be rebuilt repeatedly in each application [20]. In practice, forcing message-oriented or multi-stream designs onto TCP often reintroduces complexity that SCTP already standardizes.

C. Additional Practical Arguments

Beyond architectural correctness, several practical arguments support broader SCTP exposure:

- **Correctness:** protocol-level message framing reduces application parsing ambiguity.
- **Performance isolation:** multistreaming can reduce cross-flow head-of-line coupling.
- **Resilience:** multihoming and association semantics improve failover options.
- **Security posture:** SCTP's association handshake and extension mechanisms provide robust baseline behavior [4].

None of these claims require replacing TCP/UDP; they require giving developers a native, standards-based choice.

VIII. Limitations and Threats to Validity

This study has clear scope boundaries:

- Evaluation was performed on one Linux environment and loopback/local-host paths.
- Interop matrix validates bidirectional Go¡-¿C++ exchange, but does not exhaust all SCTP extensions.
- The current harness does not benchmark high-throughput bulk transfer against tuned TCP/UDP baselines.
- Multihoming failover scenarios were not fully exercised in a multi-interface fault-injection lab.

These limitations do not invalidate the central result (integration feasibility plus interop correctness), but they constrain generalization of quantitative performance conclusions.

Future work should include multi-host and multi-path experiments, scheduler/interleaving extension coverage, and comparative workload benchmarks for message-centric applications.

IX. Conclusion

This paper presented a Linux-focused SCTP integration into a modern Go runtime and evaluated it with in-tree tests plus Go¡-¿C++ interoperability scenarios. The evidence supports the main claim: SCTP can and should be treated as a first-class transport next to TCP and UDP in runtime networking APIs.

The broader point is architectural. Internet protocol design expects extensibility, and SCTP is a mature transport standard that addresses real application needs without forcing ad hoc reimplementation at higher layers. The implementation in this repository demonstrates that enabling SCTP in a mainstream runtime is not a disruptive redesign; it is an incremental, testable extension of existing networking abstractions.

Treating SCTP as first-class is therefore both technically justified and operationally practical.

References

[1] V. Cerf and R. Kahn, "A protocol for packet network intercommunication," *IEEE Transactions on Communications*, vol. 22, no. 5, pp. 637–648, 1974.

[2] R. Braden, "Requirements for internet hosts – communication layers," RFC Editor, RFC 1122, Oct. 1989. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1122

[3] B. Carpenter, "Architectural principles of the internet," RFC Editor, RFC 1958, Jun. 1996. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1958

[4] R. Stewart, "Stream control transmission protocol," RFC Editor, RFC 9260, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9260

[5] R. Stewart, M. Tuexen, P. Lei, and L. Ong, "Sockets api extensions for stream control transmission protocol (sctp)," RFC Editor, RFC 6458, Dec. 2011. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6458

[6] R. R. Stewart and Q. Xie, *Stream Control Transmission Protocol (SCTP): A Reference Guide*. Addison-Wesley, 2001.

[7] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad, "Stream control transmission protocol (sctp) partial reliability extension," RFC Editor, RFC 3758, May 2004. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3758

[8] R. Stewart, M. Tuexen, I. Ruengeler, and S. Loreto, "Stream schedulers and user message interleaving for the stream control transmission protocol," RFC Editor, RFC 8260, Nov. 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8260

[9] J. Postel, "Internet protocol," RFC Editor, RFC 791, Sep. 1981. [Online]. Available: https://www.rfc-editor.org/rfc/rfc791

[10] M. Kerrisk, "sctp(7) – linux manual page," https://man7.org/linux/man-pages/man7/sctp.7.html, 2025, accessed: 2026-02-14.

[11] W. Eddy, "Transmission control protocol (tcp)," RFC Editor, RFC 9293, Aug. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9293

[12] J. Postel, "User datagram protocol," RFC Editor, RFC 768, Aug. 1980. [Online]. Available: https://www.rfc-editor.org/rfc/rfc768

[13] R. Stewart, P. Lei, M. Tuexen, and E. Rescorla, "Stream control transmission protocol (sctp) dynamic address reconfiguration," RFC Editor, RFC 5061, Sep. 2007. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5061

[14] M. Tuexen, I. Ruengeler, P. Natarajan, M. Stewart, and M. Kuehlewind, "Sctp ndata: Stream schedulers and user message interleaving," RFC Editor, RFC 7053, Dec. 2013. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7053

[15] R. Jesup, S. Loreto, and M. Tuexen, "Webrtc data channels," RFC Editor, RFC 8831, Jan. 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8831

[16] ——, “Webrtc data channel establishment protocol,” RFC Editor, RFC 8832, Jan. 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8832

[17] J. R. Iyengar, P. D. Amer, and R. Stewart, “Concurrent multipath transfer using sctp multihoming over independent end-to-end paths,” *IEEE/ACM Transactions on Networking*, vol. 14, no. 5, pp. 951–964, 2006.

[18] B. Penoff, A. Wagner, M. Tuexen, and I. Ruengeler, “Portable and performant user space sctp stack,” in *Proceedings of the 21st International Conference on Computer Communications and Networks (ICCCN)*, 2012, pp. 1–9.

[19] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, “Is it still possible to extend tcp?” in *Proceedings of the 11th ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2011, pp. 181–194.

[20] J. H. Saltzer, D. P. Reed, and D. D. Clark, “End-to-end arguments in system design,” *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277–288, 1984.