Wire Formats: What They Are

Brendan Sechter

2026-01-27

A wire format is the agreed byte-level representation that two parties use when one of them has to hand structured meaning to the other and cannot hand over its memory. The name comes from the wire, the physical link between machines, but the wire is incidental. What matters is the boundary. Whenever a value must leave the address space that created it, whether it crosses a network, a file system, a process boundary, or the gap between a compiler and the machine that will execute its output, the value stops being a live structure and becomes a sequence of bytes that carries no context except what the format itself supplies. A wire format is the contract that says how those bytes are to be read back.

This article establishes what wire formats are and walks three families that are usually discussed separately. Data interchange encodings carry application values such as records, lists, and numbers. Protocol framing carries the structure of a conversation, including where one message stops and the next begins. Instruction encodings carry programs, expressed as operations a machine will perform. These three are conventionally treated as separate disciplines with separate literatures, and practitioners in one often do not think of the others as the same kind of artifact. The argument here is that they are the same kind of artifact, that the same small set of design questions arises in each, and that recognising the shared structure makes each easier to reason about. The companion article that follows this one takes the tradeoffs that cut across all three and treats them directly.

This article is descriptive rather than evaluative. It does not recommend a format, benchmark implementations, or argue that any family has solved the problem better than the others. It defines the class, establishes the properties that membership requires, and shows how each family instantiates them.

A Brief History

The problem is older than computer networking. Telegraphy already required agreement on how letters map to signals, and the Baudot code settled that question for its era in a way that later character encodings inherited. What changed with stored-program computers is that the values crossing the boundary stopped being characters and became structures, and structures admit far more representational freedom than alphabets do.

The first sustained attempt to standardise that freedom was Abstract Syntax Notation One, hereafter ASN.1, standardised by the International Telegraph and Telephone Consultative Committee and now maintained as ITU-T X.680 with encoding rules in ITU-T X.690. ASN.1 separated the abstract shape of a message from the rules that turn it into bytes, which is the single most consequential idea in the field. A schema describes what a message contains. Encoding rules describe how it is written. The same schema can be paired with different encoding rules to obtain a verbose self-describing form or a compact canonical one. Nearly every later design either adopts this separation or deliberately rejects it.

The Sun Microsystems External Data Representation standard, published as RFC 4506 and used by the Network File System, took a different position. It fixed a single encoding, aligned

everything to four-byte boundaries, and accepted the resulting padding in exchange for decoders that were trivial to write and fast to run. That tradeoff between compactness and decode simplicity recurs continuously.

The web pushed the field toward human-readable, self-describing text. The Extensible Markup Language and later JavaScript Object Notation, specified in RFC 8259, made the format inspectable with ordinary tools and removed the need to distribute a schema before two parties could talk. The cost was size and parsing expense, and the reaction against that cost produced the modern binary interchange formats, including Protocol Buffers, Apache Avro, MessagePack, and Concise Binary Object Representation, hereafter CBOR, standardised as RFC 8949. Apache Thrift arrived from the same pressure with an interface definition language attached.

Between the ASN.1 era and the web there was a decade in which remote procedure call frameworks drove the field. The Open Network Computing remote procedure call standard, RFC 5531, paired XDR with a call convention, and the Common Object Request Broker Architecture generalised the idea across languages and vendors. Those systems established that a wire format is rarely deployed alone, since it arrives attached to an interface definition language, a code generator, and a service model, and that the format is often the least contentious part of the bundle.

The web then pushed the field toward human-readable, self-describing text. The Extensible Markup Language and later JavaScript Object Notation, specified in RFC 8259, made the format inspectable with ordinary tools and removed the need to distribute a schema before two parties could talk. The cost was size and parsing expense, and the reaction against that cost produced the modern binary interchange formats, including Protocol Buffers, Apache Avro, MessagePack, and Concise Binary Object Representation, hereafter CBOR, standardised as RFC 8949. Apache Thrift arrived from the same pressure with an interface definition language attached, and BSON and Amazon Ion took the narrower path of keeping the JSON data model while replacing its byte representation.

The schema itself eventually became a specified artifact rather than an informal agreement. The Concise Data Definition Language, RFC 8610, gives CBOR a notation for describing what a valid message looks like, which closes the loop that ASN.1 opened by separating abstract syntax from encoding rules.

Instruction encodings developed on a parallel track with almost no cross-pollination. The design questions there were framed as instruction set architecture rather than serialisation, and the vocabulary differs, but the underlying problem is the same one.

Character encodings are the oldest branch of the family and the one most often forgotten. UTF-8 is a variable-length encoding of the Unicode code space, described by the Unicode Standard, and it faces exactly the questions the rest of this article treats. It is self-synchronising, which is a framing property. It is a variable-length integer encoding in all but name. It has a canonical form, and the security consequences of accepting non-canonical forms were understood early and painfully.

What Makes a Format a Wire Format

Three properties are jointly necessary. A representation that lacks any one of them is something else.

It is external. The representation is meaningful outside the process that produced it. A pointer is not a wire format because it means nothing to a reader that does not share the address space. This is why serialisation is sometimes described as the removal of context, and why the hardest part of designing a format is deciding how much context to put back.

It is agreed in advance. Both parties know how to read it before any bytes move. The agreement may be carried in a schema distributed out of band, embedded in the stream itself, or fixed by a published standard, but it exists prior to the exchange. Bytes without a prior agreement are noise.

It is unambiguous over its domain. For any byte sequence the format admits, there is exactly one value it denotes. Let B be the set of byte sequences the format accepts and V the set of values it can represent. The decode relation $D \subseteq B \times V$ must be a function, which is to say

$$\forall b \in B, \quad |\{v \in V \ : \ (b, v) \in D\}| = 1$$

A format that permits two readings of one byte sequence has failed this, and the failure is rarely benign. Where two implementations resolve the ambiguity differently, the disagreement becomes an attack surface, which is the argument Sassaman, Patterson and Bratus make in placing protocol parsing inside formal language theory. Formats that fail this admit multiple readings of the same bytes, which is a correctness problem in ordinary use and a security problem when two implementations disagree about which reading is correct.

A useful consequence follows from the second property. Because the agreement precedes the exchange, every wire format faces the question of what happens when the two parties hold different versions of the agreement. That question, treated in the companion article, is where most of the engineering difficulty lives.

Data Interchange Encodings

This family carries application values. The sender has a record, a list, a number, or a nested combination of those, and needs the receiver to reconstruct it.

The primary division within the family is whether the bytes describe themselves. A self-describing format carries type and field information inline, so a decoder can walk the stream without external help. JSON and CBOR are self-describing. A schema-dependent format carries only values, with field identity and type supplied by a schema both parties already hold. Apache Avro, specified in the Avro specification, takes this position in its pure form and writes almost nothing but the values.

Protocol Buffers, whose encoding is described in the Protocol Buffers encoding documentation, occupies a middle position that has proved durable. Each field is preceded by a tag combining a field number and a wire type. For field number f and three-bit wire type w, the tag is the varint encoding of

$$\tau = 8f + w$$

which places the wire type in the low three bits and the field number above them, so that any field numbered below sixteen yields a single-byte tag. That is why the low field numbers are worth reserving for the fields that appear most often. The field number identifies the field without naming it, which costs far less than a name, and the wire type tells a decoder how to skip a field it does not recognise even when it lacks the schema. That single decision is what makes unknown fields survivable.

The cost of self-description can be stated directly. For a message of n fields, let v_i be the encoded size of the ith value and m_i the size of its inline metadata, meaning names, type tags, or field tags. The total size is

$$S = \sum_{i=1}^{n} (m_i + v_i)$$

and the overhead fraction is

$$\phi = \frac{\sum_{i=1}^{n} m_i}{\sum_{i=1}^{n} (m_i + v_i)}$$

For JSON encoding a record of short integers, m_i includes the quoted field name, a colon, a comma, and the decimal digits are themselves larger than a binary equivalent, so ϕ frequently exceeds one half. For Protocol Buffers on the same record, m_i is typically one byte, and for Avro with an external schema m_i approaches zero. The interesting observation is not that binary is smaller, which is obvious, but that the ratio is dominated by metadata for small values and becomes irrelevant for large ones. A format carrying megabyte blobs is not meaningfully affected by its field-tag strategy.

Variable-length integer encoding illustrates how representation choices interact with data distribution. In the base-128 varint scheme used by Protocol Buffers, seven bits of payload travel in each byte and the eighth signals continuation, so a non-negative integer x occupies

$$L(x) = \max\left(1, \left\lceil \frac{\lfloor \log_2 x \rfloor + 1}{7} \right\rceil\right)$$

bytes. Small numbers become short, which is the intent, but a value near 2^{64} takes ten bytes where a fixed-width encoding would take eight. The bet pays only below a threshold. A varint is strictly smaller than a fixed encoding of W bytes when $L(x) < W$, and for $W = 8$ that condition is

$$x < 2^{49}$$

since a value of fifty bits or more already requires eight varint bytes. Varints are a bet that the distribution is concentrated near zero, and like any bet they can be lost.

Zero-copy formats such as Cap’n Proto and FlatBuffers take a further step and arrange the encoded bytes so that they can be read in place without a decode pass. The Cap’n Proto encoding specification describes the layout. Access becomes pointer arithmetic over a buffer rather than construction of new objects, which changes the cost model substantially, and the price is a less compact representation and a stricter layout contract.

Two further positions are worth naming because they show the ends of the range. Simple Binary Encoding, developed for electronic trading, fixes field offsets at compile time so that access is a constant-offset load with no parsing at all, accepting a rigid layout in exchange for latency that can be reasoned about deterministically. At the opposite end, the encodings used by blockchain systems optimise for agreement rather than speed. Borsh, used in the Solana ecosystem, is deliberately minimal and deterministic, and Ethereum’s recursive length prefix encoding describes only the structure of nested byte strings and leaves all typing to the application. Neither is trying to be fast in the sense the trading formats mean. Both are trying to be unambiguous in the strong sense the companion article calls canonicity.

The family therefore spans a wide range of positions on a small number of axes. A format may be text or binary, self-describing or schema-dependent, parsed into new objects or read

in place, and permissive or canonical. Most of the named formats are simply particular combinations of those four choices, and the combinations that are absent are usually absent because they are contradictory rather than because nobody has tried them.

Protocol Framing

This family carries the structure of a conversation rather than the values within it. Its central question is not how to represent an integer but where a message ends.

A byte stream such as a Transmission Control Protocol connection delivers ordered bytes with no record boundaries. Any structure above that must be imposed by the format. Three strategies dominate. Length prefixing writes the size before the payload, which lets a reader allocate once and know exactly when it is finished. Delimiting reserves a byte sequence to mark the end, which is simple but requires escaping any occurrence of the delimiter inside the payload. Self-describing framing derives the extent from the content itself, as a decoder does when it reads a nested structure and knows it is complete.

The HyperText Transfer Protocol shows both the older and newer approaches. Version 1.1 uses delimiters for headers, terminating them with a blank line, and a length prefix for bodies via the content-length header, with chunked transfer encoding as the fallback when the length is not known in advance. Version 2, specified in RFC 9113, abandons the text framing entirely for fixed nine-octet binary frame headers carrying length, type, flags, and stream identifier. That change was made to allow many logical streams to share one connection, which text framing cannot express without ambiguity.

Header compression illustrates that framing formats face the same metadata pressure as interchange formats. HTTP/2 introduced HPACK, which maintains a shared table of previously seen header fields so that a repeated header can be sent as a small index rather than a full name and value. This is the same insight as Avro’s external schema, applied to a conversation instead of a record, and arrived at independently.

Framing overhead is a ratio worth stating. For a frame carrying payload of size p with header of size h, the useful fraction is

$$\eta = \frac{p}{p + h}$$

Framing also nests, and each layer of encapsulation charges its own header. For n layers with header sizes h_1 through h_n, the useful fraction is

$$\eta_{\text{stack}} = \frac{p}{p + \sum_{i=1}^{n} h_i}$$

so a bundle inside a transport segment inside a network packet pays all three, and a small payload can carry more framing than content once the stack is deep enough. For an HTTP/2 frame with $h = 9$ carrying a kilobyte, η exceeds ninety nine percent and the header is irrelevant. For the same header carrying a four-byte acknowledgement, η falls below one third. Protocols that exchange many small messages therefore care enormously about header size, which is why QUIC, specified in RFC 9000, works to keep its packet headers short and why it moved framing into the encrypted payload to permit change without ossification.

The framing question recurs at every layer of a protocol stack, and the answers differ layer by layer. An Ethernet frame is length-delimited by the physical layer. The Internet Protocol header carries an explicit total length, as does its version 6 successor in a different field layout. The User Datagram Protocol preserves message boundaries and so needs no framing

above it, which is exactly why protocols that want records often prefer it. The Transmission Control Protocol destroys those boundaries and forces every protocol above it to reinvent framing, which is the single most consequential framing decision in the history of the field. The Stream Control Transmission Protocol was designed in part to give back the message boundaries that TCP removes.

Above the transport, the Transport Layer Security record layer imposes its own framing, and the application protocols impose theirs again. The WebSocket Protocol adds a small binary frame header to a connection that began as HTTP, HTTP/3 carries HTTP semantics over QUIC streams rather than over TCP, and messaging protocols such as MQTT and AMQP define their own compact frame headers for the publish-and-subscribe case. gRPC layers a length-prefixed message framing over HTTP/2 streams, which means a single gRPC call is framed at least four times before it reaches the wire.

Delay-tolerant networking pushes framing into an environment where the round trip may be hours. The Bundle Protocol, now RFC 9171, frames application data into bundles that carry enough context to be forwarded and stored by intermediaries that may hold them for a long time before onward transmission. The architecture is described in RFC 4838, and the corpus covers the practical side of this in Getting Started with ION-DTN on FreeBSD, with bundle-level exchange demonstrated in Almost Serving a Web Page with ION-DTN bpchat and Serving a Web Page with ION-DTN bpsendfile and bprecvfile. When storage time is measured in hours, a bundle is closer to a self-contained document than to a packet, and its framing reflects that.

Instruction Encodings

This family carries programs. An instruction encoding is the byte-level representation of operations that a machine, physical or virtual, will execute.

Practitioners rarely call these wire formats, and the omission is instructive. An instruction stream satisfies all three membership properties. It is external, since the processor did not create it and holds no context from the compiler. It is agreed in advance, by the architecture specification. It is unambiguous, or the machine could not execute it. The reason instruction encodings are not usually grouped with the others is historical rather than principled.

The design questions match the interchange family closely. Fixed-width encodings, such as the base RISC-V integer instruction set described in the RISC-V unprivileged specification, make decoding trivial because the next instruction is always a fixed distance away, at the cost of density. Variable-width encodings achieve better density and complicate decoding, which is the same tradeoff varints make against fixed-width integers, and the compressed RISC-V extension adds sixteen-bit forms for exactly the density reason. Practical exposure to the fixed-width case appears in the corpus in UNIX ARM Assembler on Android.

Instruction density can be treated the same way as encoding overhead. For a program of k instructions with encoded sizes c_j, the mean instruction size is

$$\bar{c} = \frac{1}{k} \sum_{j=1}^{k} c_j$$

and the total text size is $k\bar{c}$. If a compressed encoding renders a fraction ρ of instructions at half width, the mean becomes

$$\bar{c}_{\text{compressed}} = \bar{c} \left(1 - \frac{\rho}{2}\right)$$

so compressing half the instruction stream saves a quarter of the text. A compressed encoding reduces $\bar{c}$, which matters when instruction memory is the binding constraint, as it is on the embedded targets discussed in Getting Started with no_std Rust Programming and no_std Rust with bin and lib.

Hardware encodings occupy a spectrum. The variable-length x86 encoding grew by accretion across four decades and admits instructions from one to fifteen bytes, which buys density and imposes a decoding problem so difficult that implementations cache decoded forms rather than repeat the work. The ARM architecture took the fixed-width position for its original instruction set, then added the Thumb encodings to recover density for memory-constrained parts, arriving at the same compromise RISC-V later reached with its compressed extension. The pattern across all three is that fixed width is chosen first for decoder simplicity and variable width is added later under density pressure, which is the same sequence the interchange family followed from XDR to the modern binary formats.

Virtual machine encodings make the wire-format character explicit, because the bytes genuinely travel. WebAssembly, specified by the W3C WebAssembly Core Specification and introduced by Haas and colleagues, is a binary instruction encoding designed to be transmitted over a network, validated on arrival, and executed. It carries an explicit type section so that a consumer can check the module before running it, which is precisely the schema-versus-self-description question in another guise. The corpus demonstrates delivering such a module in WASM on a Jekyll Blog with Rust and wasm-bindgen.

A bytecode designed for verification makes the wire-format character explicit in a different way. The Keleusma project treats its bytecode as a versioned artifact with a stated compatibility boundary rather than as an implementation detail, described in Keleusma’s Self-Hosting Strategy, with the surface language introduced in Getting Started with Keleusma 0.1.1. Attaching a version number to an instruction stream is the same discipline the interchange family arrived at independently, reached from the opposite direction.

Managed-runtime bytecodes show the design space clearly because they were specified rather than accreted. The Java Virtual Machine class file format is a stack-oriented encoding with a constant pool, so that repeated references become short indices, which is structurally the same move HPACK makes for headers and Avro makes for schemas. The Common Intermediate Language standardised as ECMA-335 takes a comparable position for a different runtime. The Dalvik executable format deliberately chose a register machine over a stack machine and shared its constant pools across an entire application, trading a more complex format for smaller total size on devices where storage was the binding constraint. LLVM bitcode is a wire format for compiler intermediate representation rather than for execution, and it carries an explicit versioning scheme because it must survive between tool versions.

The kernel extension case is the most recent addition. The Berkeley Packet Filter instruction set, now standardised as RFC 9669, is an encoding whose consumer verifies the program before running it, in an environment where a mistake compromises the kernel. Verification-before-execution is the same posture WebAssembly adopts, reached independently under a different threat model, and it makes explicit that an instruction encoding is a message from an untrusted party until proven otherwise.

Blockchain instruction encodings share the same shape with an additional constraint. A transaction is an instruction stream that must be interpreted identically by every participant, since disagreement is a consensus failure rather than a bug in one implementation. That requirement forces canonical encoding, which the companion article treats as a general concern. The Ethereum Virtual Machine encoding and the Solana variant of the Berkeley Packet Filter encoding both sit in this category, and the corpus covers the practical side in Getting Started with Solana Using Rust and Anchor.

What the Three Families Share

Set side by side, the same questions recur.

Every family must decide where the schema lives, whether inline with the data, distributed out of band, or fixed by a standard. Every family must decide how a reader knows where a unit ends, whether by length prefix, delimiter, fixed width, or derivation from content. Every family must decide what a reader does with something it does not recognise, whether it skips it, fails, or refuses to proceed. Every family must decide how much it will pay in bytes for the ability to inspect the representation without special tooling. Every family must decide whether two encoders given the same value are required to produce the same bytes.

How much structure a format exposes to a machine rather than to a human is the same question the hypermedia literature asks of markup, where link typing and machine-readable structure are exactly the axes on which a format either supports automated processing or forces the reader to guess. That comparison is developed in Deficiencies of the HTML Hypermedia Model.

The vocabulary differs. An interchange format has fields and a schema, a protocol has frames and a specification, an architecture has instructions and an instruction set manual. The decisions are the same decisions, and an engineer who has made them carefully in one family has most of what is needed to reason about another.

Epistemic State

The classification into three families is an organising choice made by this article, not a standard taxonomy. Other divisions are defensible, including splitting interchange formats by self-description or grouping by whether the format targets humans. Readers should treat the three-way split as a useful lens rather than a claim about how the field is actually organised.

The claim that instruction encodings satisfy the definition of a wire format is an argument, not a reported consensus. The literature does not usually group them this way. The argument rests on the three membership properties, and a reader who rejects those properties as the right definition will reasonably reject the grouping.

The survey aims for breadth across the three families rather than depth in any one, and it is not exhaustive. Formats are named to illustrate a position on a design axis, so a format's absence carries no judgment, and several significant designs are omitted for space rather than for lack of merit. Where a format is characterised in one sentence, that sentence is a summary of its published specification and not an evaluation.

The historical account is compressed and selective. ASN.1, XDR, JSON, and the binary interchange formats are treated as a sequence, which understates how much development ran in parallel and omits formats that were significant in their contexts. The claim that the abstract-syntax and encoding-rules separation is the field's most consequential idea is a judgment.

The equations are definitional rather than empirical. They state how overhead, varint length, framing efficiency, and mean instruction size are computed. They are not measurements, and no benchmark in this article supports a claim that one format outperforms another on real workloads.

Statements about specific formats are drawn from their published specifications, which are cited. Where a specification has been revised, the current version is cited even when the design decision being described originated in an earlier one.

Out of Scope

Character encodings are treated only in passing. The mapping from characters to bytes is a genuine wire-format problem with a substantial literature, and it deserves its own treatment.

Compression is excluded. Every format discussed can be compressed by a general-purpose algorithm, and the interaction between format design and compressibility is real, but it is a separate subject.

Cryptographic concerns are excluded except where canonical encoding is mentioned. Authenticated encryption, signature schemes, and the ways a format can undermine them are out of scope here.

Remote procedure call frameworks are excluded. They compose a wire format with a service model, and the service model is a different topic.

Database storage formats and columnar file formats are excluded. They share ancestry with interchange encodings but optimise for query rather than transmission.

No benchmarks appear. This article makes no performance claims about any implementation.

The tradeoffs that cut across the three families are the subject of the second article in this series and are deliberately not resolved here.

Conclusion

A wire format is the byte-level agreement that lets structured meaning cross a boundary. Membership requires that the representation be external, agreed in advance, and unambiguous. Three families that are usually discussed apart all satisfy the definition. Data interchange encodings carry values, protocol framing carries conversation structure, and instruction encodings carry programs.

The families were developed by different communities with different vocabularies, and the separation has obscured how much they share. Each must decide where the schema lives, how a reader finds the end of a unit, what happens on encountering the unrecognised, how much to pay for inspectability, and whether encoding must be canonical. Those five questions are the subject of the companion article, which treats them as one set of tradeoffs rather than three.

References

- Hennessy, John L. and Patterson, David A., Computer Architecture, A Quantitative Approach, Morgan Kaufmann
- Stevens, W. Richard, TCP/IP Illustrated, Volume 1, The Protocols, Addison-Wesley
- Tanenbaum, Andrew S. and Wetherall, David J., Computer Networks, Pearson
- Apache Avro Specification
- Amazon Ion Binary Encoding
- Apache Thrift Interface Definition Language
- BSON Specification
- Baudot Code
- Borsh Specification
- Cap’n Proto Encoding Specification
- Dalvik Bytecode Format
- ECMA-335, Common Language Infrastructure
- Ethereum Recursive Length Prefix Encoding
- FlatBuffers Documentation
- IEEE 802.3, Ethernet

- Java Virtual Machine Specification, The class File Format
- LLVM Bitcode File Format
- ITU-T X.680, Abstract Syntax Notation One, Specification of Basic Notation
- ITU-T X.690, ASN.1 Encoding Rules, BER, CER and DER
- MQTT Specification
- MessagePack Specification
- OASIS AMQP Version 1.0
- Protocol Buffers Encoding Documentation
- RFC 768, User Datagram Protocol
- RFC 791, Internet Protocol
- RFC 1813, NFS Version 3 Protocol Specification
- RFC 3629, UTF-8, a Transformation Format of ISO 10646
- RFC 4838, Delay-Tolerant Networking Architecture
- RFC 4506, XDR, External Data Representation Standard
- RFC 5531, Remote Procedure Call Protocol Version 2
- RFC 7541, HPACK, Header Compression for HTTP/2
- RFC 6455, The WebSocket Protocol
- RFC 8259, The JavaScript Object Notation Data Interchange Format
- RFC 8200, Internet Protocol Version 6
- RFC 8446, Transport Layer Security Version 1.3
- RFC 8610, Concise Data Definition Language
- RFC 8949, Concise Binary Object Representation
- RFC 9000, QUIC, A UDP-Based Multiplexed and Secure Transport
- RFC 9112, HTTP/1.1
- RFC 9113, HTTP/2
- RFC 9171, Bundle Protocol Version 7
- RFC 9114, HTTP/3
- RFC 9260, Stream Control Transmission Protocol
- RFC 9293, Transmission Control Protocol
- RFC 9669, BPF Instruction Set Architecture
- RISC-V Unprivileged Specification
- Apache Thrift Interface Definition Language
- Simple Binary Encoding
- Unicode Standard
- W3C Extensible Markup Language
- gRPC Core Concepts
- W3C WebAssembly Core Specification
- Haas, Andreas et al., Bringing the Web up to Speed with WebAssembly, PLDI, 2017
- Sassaman, Len, Patterson, Meredith L. and Bratus, Sergey, Security Applications of Formal Language Theory, IEEE Systems Journal, 2013
- Related Post, Almost Serving a Web Page with ION-DTN bpchat
- Related Post, Deficiencies of the HTML Hypermedia Model
- Related Post, Getting Started with ION-DTN 3.4.0 on FreeBSD
- Related Post, Getting Started with Keleusma 0.1.1
- Related Post, Getting Started with Solana Using Rust and Anchor
- Related Post, Getting Started with no_std Rust Programming
- Related Post, Keleusma’s Self-Hosting Strategy
- Related Post, Serving a Web Page with ION-DTN bpsendfile and bprecvfile
- Related Post, UNIX ARM Assembler on Android
- Related Post, WASM on a Jekyll Blog with Rust and wasm-bindgen
- Related Post, no_std Rust with bin and lib