

arf - Another RPC Framework

Vito Sartori, November 2025

Abstract

The Another RPC Framework (stylized **arf**) defines an interface description language (IDL), a compact binary serialization format, and a request/response protocol for service-oriented communication.

arf aims to provide a simple, evolvable, and highly efficient system for defining structs, services, and RPC methods while eliminating the need for field numeric identifiers in IDL source files. arf emphasizes readability, forward compatibility, and minimal over-the-wire footprint.

1 Status of This Memo

This document is an Internet-Draft and is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). They are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted at any time.

2 Copyright Notice

Copyright © 2025 Vito Sartori. All rights reserved.

Contents

1	Status of This Memo	2
2	Copyright Notice	2
3	Introduction	6
4	Conventions and Terminology	6
4.0.1	Terms	6
5	The arf Language	7
5.1	Packages	7
5.2	Imports	8
5.3	Types	8
5.4	Struct Declarations	9
5.5	Service Declarations	10
5.5.1	Re-Openable Services	10
5.5.2	Methods	11
5.5.3	IDL Syntax for Streams	12
5.6	Fully-Qualified Names	12
6	Serialization	13
6.1	Overview	13
6.2	Length Encoding	13
6.3	Signed Integer Encoding	13
6.3.1	ZigZag Transformation	14
6.3.2	Width Enforcement	14
6.3.3	Rationale	15
6.3.4	Worked Examples	15
6.3.5	Mapping Summary (Signed \rightarrow ZigZag \rightarrow VarUInt)	15
6.4	Primitive Types	16
6.5	Composite Types	16
6.5.1	Arrays	17
6.5.2	Maps	17
6.5.3	Structs	17
6.6	Optional Fields	18
6.7	Error Handling	18
6.8	Binary Layout Example	19
7	RPC Protocol	20
7.1	Frame Format	20
7.1.0.1	MessageKind	20
7.2	Transport Assumptions	21
7.3	Connection Model and Multiplexing	21

7.3.1	Correlation Identifiers	21
7.3.2	Concurrent RPCs	22
7.3.3	Ordering Guarantees	22
7.4	Identifiers	22
7.4.1	FNV-1a Hash Function	23
7.5	Requests	24
7.6	Method Forms and Allowed Frame Sequences	24
7.6.1	RPC Completion	25
7.7	Interleaving of Input and Output Streams	26
7.8	Unary vs Streaming Semantics	26
7.8.1	Unary Input	26
7.8.2	Input Stream	26
7.8.3	Unary Output	27
7.8.4	Unary Output Ordering	27
7.8.5	Output Stream	27
7.9	Per-form Examples	28
7.9.1	Form: NNNN (no input, no output, no streams)	28
7.9.2	Form: NNNY (server-streaming only)	28
7.9.3	Form: YNYN (unary input, input stream, no unary output)	28
7.9.4	Form: YYYY (full bidirectional streaming + unary input/output)	28
7.10	Cancellation and Error Semantics	29
7.11	Backpressure	29
8	Evolution and Compatibility	29
8.1	Adding Fields	30
8.2	Removing Fields	30
8.3	Changing Types	30
8.4	Versioning Through Packages	31
9	ABNF Grammar	31
10	Security Considerations	34
11	IANA Considerations	35
12	References	35
13	Acknowledgements	35
14	Appendices	36
A	FNV-1a-32 Reference Implementation	36
B	FNV-1a-32 Test Vectors (Informative)	36

C	C Reference Implementation for ZigZag Encoding	37
D	Test Vectors for Signed Integer Encoding	38
D.1	Common Values	38
D.2	Boundary Values	38
D.2.1	int8	38
D.2.2	int16	38
D.2.3	int32	38
D.2.4	int64	39
E	Implementation Notes (Informative)	39
E.1	Recommended Limits	39
E.2	VarUInt and Signed Integer Decoding	40
E.3	Framing and Buffering Strategies	40
E.4	Stream and RPC Lifecycle Management	41
E.5	Timestamps and Clock Handling	41
E.6	Identifier Caching	42
E.7	Error Mapping and Diagnostics	42

3 Introduction

arf (Another RPC Framework) is a compact, binary, schema-first RPC mechanism designed for service-oriented systems. arf provides:

- A simple IDL with packages, imports, structs, and services.
- Efficient binary encoding without field tags on the wire.
- Optional types without nested “option” wrappers.
- Evolvability through package-level versioning.
- Re-openable service definitions for modular organization.

arf draws inspiration from Protobuf, Cap’n Proto, and Thrift while focusing on ergonomics and avoiding field-ID management clutter.

4 Conventions and Terminology

The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHALL**, **SHALL NOT**, **SHOULD**, **SHOULD NOT**, **RECOMMENDED**, **NOT RECOMMENDED**, **MAY**, and **OPTIONAL** in this document are to be interpreted as described in [RFC2119] and [RFC8174] when, and only when, they appear in all capitals.

4.0.1 Terms

Client The endpoint that initiates an RPC by sending an **INVOKE** frame.

Server The endpoint that receives an **INVOKE** frame and executes the corresponding method.

Endpoint Either the client or the server participating in an RPC.

RPC A single remote procedure call identified by one **CorrelationID** and consisting of an **INVOKE** frame and any subsequent frames that share that identifier.

Frame A discrete protocol message sent over the underlying transport representing an invocation, stream element, or control signal, as defined in Section 7.1.

CorrelationID The identifier that associates all frames belonging to the same RPC on a given connection.

Active RPC An RPC for which an **INVOKE** frame has been sent and for which the RPC completion conditions defined in Section 7.6.1 have not yet been satisfied.

Unary A non-streaming RPC parameter or result, serialized as a single value rather than as a sequence.

Stream A potentially unbounded sequence of values of a given type, carried using `IN_STREAM/IN_CLOSE` or `OUT_STREAM/OUT_CLOSE` frames.

Input stream A stream of values flowing from client to server as part of an RPC.

Output stream A stream of values flowing from server to client as part of an RPC.

Form One of the sixteen method signature combinations defined in Section 7.6, derived from the presence or absence of unary input, unary output, input stream, and output stream.

Struct A user-defined composite type consisting of a fixed sequence of named fields.

Field A named component of a struct, identified for serialization by its ordinal position within the struct declaration.

Optional field A struct field whose type is `optional<T>`, whose presence or absence is indicated by the struct's presence bitmap.

Package A named collection of type and service declarations that forms a versioned namespace boundary, as defined in Section 5.1.

Service A named collection of RPC methods defined within a package.

Method A single RPC operation defined within a service, identified by its name and method form.

VarUInt The variable-length unsigned integer encoding used for lengths and similar quantities, as defined in Section 6.2.

Unix Epoch The time origin defined as 1970-01-01T00:00:00Z in Coordinated Universal Time (UTC), from which arf timestamps measure elapsed milliseconds.

5 The arf Language

5.1 Packages

An arf source file **MUST** begin with a **package** declaration:

Package names establish namespace boundaries and versioning domains. A package name uniquely identifies all types and services defined within the file. Each type defined in a package is assigned a fully-qualified name of the form:

```
1 package vibeta1.myservice;
```

```
<package-name> "." <type-name>
```

Example:

```
vibeta1.mypackage.OtherName
```

5.2 Imports

A source file MAY include one or more **import** directives that logically incorporate definitions from other arf source files:

```
1 import "../..common/vibeta1/common";
```

Each imported file MUST contain a **package** declaration. The package name of the imported file determines the fully-qualified names of the types it defines.

An import MAY specify an explicit local alias using the **as** keyword. If no alias is provided, the compiler MUST derive an implicit alias from the final component of the imported package name.

All aliases within a single source file MUST be unique. If two imports would produce the same implicit alias, or if an implicit alias collides with an explicit alias, the compiler MUST reject the source file.

Referenced types of the form **<alias>.<TypeName>** MUST resolve unambiguously to exactly one imported package. Fully-qualified names MAY be used instead of aliases and MUST NOT depend on import aliasing.

5.3 Types

arf provides a set of builtin types that form the foundation of the serialization model. These types MAY appear as struct fields and as components of composite types. Builtin primitive types MUST NOT appear directly as unary method parameters, unary method results, or stream element types.

- **bool**: a boolean value.
- **int8, int16, int32, int64**: signed integers of the specified width.
- **uint8, uint16, uint32, uint64**: unsigned integers of the specified width.
- **float32, float64**: IEEE 754 binary floating-point values.
- **string**: a sequence of UTF-8 encoded Unicode scalar values.
- **timestamp**: represents an instant on the UTC time-line encoded as an unsigned integer count of milliseconds since the UTC Unix Epoch.
- **bytes**: an arbitrary sequence of octets.
- **array<T>**: an ordered, variable-length sequence of values of type T.
- **map<K, V>**: an unordered, variable-length association between keys of type K and values of type V.
- **optional<T>**: a value of type T that MAY be present or absent.

User-defined **struct** types MAY also be declared within a package. All types, whether builtin or user-defined, participate uniformly in the serialization rules described in Section 6. Permitted map key types are:

- any unsigned or signed integer type, and
- any **enum** type.

All other types are forbidden as map keys.

Enums are encoded as **uint16** representing the zero-based ordinal position of the enumeration value in the declaration. Enumeration values MUST NOT exceed the range of **uint16**.

5.4 Struct Declarations

Structs define a sequence of fields, each consisting of a name and a type, in the order in which they are written:

```
1 struct TeamNameRequest {
2     name string;
3 }
```

Field numbers **MUST NOT** appear in arf source files. Instead, each field is assigned an implicit ordinal index based solely on its position in the declaration, beginning at zero for the first field. Ordinal indices **MUST** remain stable within a package version, as they are referenced by the binary serialization rules defined in Section 6.

Struct declarations **MAY** include nested struct definitions. Nested structs are assigned fully-qualified names in the same manner as top-level structs.

5.5 Service Declarations

Services define RPC methods:

```

1 service NameService {
2     check_name(req CheckNameRequest) -> common.
3     CheckNameResponse;
4 }

```

Method overloading (multiple methods with the same name but different signatures) is not permitted. Method names **MUST** be unique within a service.

Each method name **MUST** be unique within the merged logical service definition formed after processing all re-opened service blocks. Multiple declarations of the same method name are permitted only when their signatures are identical.

5.5.1 Re-Openable Services

A service **MAY** be declared in multiple **service** blocks within the same package. Re-opening a service does not introduce independent scopes. All method declarations across blocks participate in a single unified namespace:

```

1 service NameService {
2     check_name(req CheckNameRequest) -> common.
3     CheckNameResponse;
4 }
5
6 service NameService {
7     check_slug(req CheckSlugRequest) -> common.
8     CheckSlugResponse;
9 }

```

All **service** blocks with the same service name **MUST** be merged into a single logical service definition during compilation. If a method name appears in more

than one block, the compiler MUST determine whether the corresponding method signatures are identical.

Two method declarations are considered to have *identical signatures* if and only if all of the following hold:

- They declare the same method name.
- They declare the same ordered sequence of unary input parameters.
- They declare the same input stream type (or both omit it).
- They declare the same ordered sequence of unary output values.
- They declare the same output stream type (or both omit it).

If any of these conditions is violated, the signatures are divergent, and the compiler MUST reject the program.

When a service is declared across multiple **service** blocks, all method names contribute to the same fully-qualified method namespace. Duplicate method names across service blocks do not constitute overloading; they represent re-declaration and MUST be byte-for-byte equivalent in signature.

5.5.2 Methods

An arf method signature consists of zero or more unary input parameters, zero or more unary output parameters, and at most one input stream and one output stream. The presence or absence of each of these components determines a *method form*. Four independent boolean dimensions are defined:

- **HasInput**: whether the method declares at least one unary input parameter.
- **HasOutput**: whether the method declares at least one unary output value.
- **HasInputStream**: whether the method declares a single input stream parameter.
- **HasOutputStream**: whether the method declares a single output stream.

The combination of these four booleans yields sixteen method forms. Each form has a well-defined set of legal frame sequences, as specified in Section 7.6.

All unary parameters, unary results, and stream element types MUST be defined as either **struct** or **enum** types. Primitive and composite types MUST NOT appear directly in RPC method signatures. This limitation was deliberately created to ensure all data transmitted through RPC methods are part of structures, enforcing all messages to be explicitly defined.

5.5.3 IDL Syntax for Streams

Streams are declared using the **stream** keyword in either the input parameter list or the output tuple. A method MAY declare at most one input stream and at most one output stream.

Examples:

```

1 NNYN(stream T);
2 NNYN() -> (0, stream U);
3 YYYY(i I, stream T) -> (0, stream U);

```

A streaming parameter is written as **stream** <Type>. Unary parameters are written as standard name-type bindings. Streams MUST NOT appear more than once on either side of the signature.

5.6 Fully-Qualified Names

Every type and service defined within a package is assigned a fully-qualified name of the form:

```
<package-name> "." <identifier>
```

Fully-qualified names uniquely identify declarations across all packages. Two declarations with distinct fully-qualified names are considered distinct, even if their unqualified names are identical or their definitions are structurally equivalent.

Within a source file, references to types defined in other packages MAY appear in either of the following forms:

- **Aliased form:** <alias>.<TypeName>, where <alias> is the explicit or implicit import alias established by an **import** directive; or
- **Fully-qualified form:** <package-name>.<TypeName>.

Aliased references MUST resolve unambiguously to exactly one imported package. Fully-qualified references MUST NOT depend on import aliasing and MUST resolve solely based on the declared package name.

Fully-qualified names MUST remain stable across all files belonging to the same package version. Changing the package name of any declaration constitutes a breaking change and therefore requires introducing a new package version.

Methods are assigned fully-qualified names of the form:

```
<package-name> "." <service-name> "." <method-name>
```

This string is the canonical method identity used for identifier derivation.

6 Serialization

6.1 Overview

arf defines a compact binary serialization format for values of all builtin and user-defined types. Serialization is deterministic and does not depend on field names or field numbers appearing in the IDL. Struct fields are serialized in declaration order using implicit ordinal indices as described in Section 5.4.

All application-level integer *values* (such as field values and lengths) are encoded using the **VarUInt** scheme (and ZigZag + **VarUInt** for signed integers), unless explicitly stated otherwise. Floating-point values are encoded in IEEE 754 binary formats using little-endian byte order. Variable-length entities (such as strings, byte sequences, arrays, and maps) are prefixed with a length encoded using the variable-length integer scheme described in Section 6.2.

All serialized representations are self-delimiting and do not require out-of-band framing. The RPC protocol defines its own framing mechanism for transporting serialized values.

6.2 Length Encoding

arf uses an unsigned variable-length integer format (referred to in this document as *VarUInt*) for encoding lengths of variable-sized objects. **VarUInt** uses a base-128 continuation-bit scheme:

- Each byte contributes seven bits of payload and one continuation bit.
- For all but the final byte, the most-significant bit (MSB) MUST be set to 1.
- The final byte MUST have its MSB set to 0.

This format allows lengths to be encoded using between one and ten bytes, depending on magnitude. An implementation MUST NOT accept lengths that exceed the representational capacity of a 64-bit unsigned integer.

Unless explicitly stated, all length-prefixed types in this section use **VarUInt** for length encoding.

6.3 Signed Integer Encoding

Signed integer values (**int8**, **int16**, **int32**, **int64**) are encoded using a two-step process:

1. The signed value is transformed into an unsigned integer using ZigZag encoding.
2. The resulting unsigned integer is serialized using the **VarUInt** scheme defined in Section 6.2.

6.3.1 ZigZag Transformation

ZigZag encoding maps signed integers to unsigned integers such that small magnitude values (both positive and negative) produce small unsigned values when serialized. This ensures that numerically small signed values compress efficiently under **VarUInt** encoding.

For a signed integer value n of bit-width w , the ZigZag transform is defined as:

$$z = (n \ll 1) \oplus (n \gg (w - 1))$$

Where:

- \ll denotes arithmetic left-shift,
- \gg denotes arithmetic right-shift,
- \oplus denotes bitwise exclusive OR, and
- w is the width of the signed integer type in bits.

The inverse transformation is defined as:

$$n = (z \gg 1) \oplus -(z \wedge 1)$$

6.3.2 Width Enforcement

Although ZigZag and **VarUInt** operate on unbounded integers in theory, arf enforces strict width limits based on the declared type.

Decoders *MUST* reject any decoded value whose magnitude exceeds the range of the declared signed type:

- **int8**: $-128 \leq n \leq 127$
- **int16**: $-32,768 \leq n \leq 32,767$
- **int32**: $-2^{31} \leq n \leq 2^{31} - 1$
- **int64**: $-2^{63} \leq n \leq 2^{63} - 1$

Encoders *MUST NOT* emit values outside the representable range of the declared type.

6.3.3 Rationale

ZigZag encoding ensures that:

- small negative integers do not expand to large encodings,
- signed and unsigned integers share a common storage format,
- the encoding is independent of machine endianness, and
- integer encodings remain stable across platforms and languages.

This scheme enables compact representation of signed integers while preserving fast decoding and deterministic serialization.

6.3.4 Worked Examples

The following examples illustrate ZigZag transformation and the resulting VarUInt encoding (32-bit width shown for clarity):

Signed	ZigZag(n)	Decimal	VarUInt Bytes
0	0x00	0	00
-1	0x01	1	01
1	0x02	2	02
-2	0x03	3	03
2	0x04	4	04
300	0x0258	600	D8 04
-300	0x0257	599	D7 04

For example, for $n = -1$:

$$z = (-1 \ll 1) \oplus (-1 \gg 31) = 1$$

ZigZag(-1) therefore produces 0x01, which is encoded as a single-byte VarUInt.

6.3.5 Mapping Summary (Signed \rightarrow ZigZag \rightarrow VarUInt)

Signed	ZigZag(n)	VarUInt Encoding
0	0	00
-1	1	01
1	2	02
-2	3	03
2	4	04
-3	5	05
3	6	06
300	600	D8 04
-300	599	D7 04

6.4 Primitive Types

The serialization of primitive types is defined as follows:

- **bool**: encoded as a single byte. The value `0x00` represents **false**; the value `0x01` represents **true**. No other values are permitted.
- **uint8, uint16, uint32, uint64**: encoded as `VarUInt` (Section 6.2). The encoded value MUST lie within the range of the corresponding unsigned integer type. Decoders MUST reject values that exceed the representable range of the declared type.
- **int8, int16, int32, int64**: encoded as a signed variable-length integer using ZigZag encoding followed by `VarUInt`. Let n be the signed value and z the ZigZag-transformed unsigned integer. Encoders MUST compute z and emit it using `VarUInt`; decoders MUST recover n from z and verify that it lies within the range of the declared signed type.
- **float32, float64**: encoded in IEEE 754 binary32 or binary64 format respectively, using little-endian byte order.
- **string**: encoded as `VarUInt(length)` followed by `length` bytes containing UTF-8 encoded scalar values. Invalid UTF-8 byte sequences MUST cause deserialization to fail.
- **bytes**: encoded as `VarUInt(length)` followed by `length` raw octets.
- **timestamp**: encoded as `VarUInt` representing the number of milliseconds elapsed since the UTC Unix Epoch. Implementations that cannot represent millisecond precision MUST truncate toward zero.

These encodings are used uniformly whether a primitive appears as a struct field or within a composite type. However, primitives MUST NOT be used as unary method parameters, unary method results, or as stream elements. For methods, unary inputs, unary outputs, and streams MUST use structs or enums as specified in Section 5.5.2.

Key equality is defined by equality of the serialized key value.

For integer keys, numeric equality applies. For enumeration keys, ordinal equality applies.

Because structured types are forbidden as keys, equality testing is deterministic and implementation-independent.

6.5 Composite Types

Composite types define container-like structures with their own encoding rules.

6.5.1 Arrays

An `array<T>` value is encoded as:

1. `VarUInt(n)`, where n is the number of elements.
2. The concatenation of the serialized representations of the n elements in order.

An array MAY be empty. Empty arrays are encoded as `VarUInt(0)` followed by zero bytes.

6.5.2 Maps

A `map<K, V>` value is encoded as:

1. `VarUInt(n)`, where n is the number of key-value pairs.
2. For each pair, the serialized key immediately followed by the serialized value.

A map MAY be empty. Map iteration order MUST be the order chosen by the encoder and MUST be preserved by the decoder without implying semantic ordering or sorting.

Keys MUST be unique according to the equality semantics of type `K`. Deserializers MUST reject maps that contain duplicate keys.

6.5.3 Structs

Struct values are serialized as *length-prefixed records*. Each struct instance is preceded by a byte length that defines the total size of the struct body in bytes, allowing decoders to skip unknown fields safely.

A struct value is encoded as:

1. `VarUInt(L)`, where L is the number of bytes in the struct body that follows; and
2. the struct body, consisting of the presence bitmap and serialized field values as described below.

No field numbers or tag identifiers appear on the wire; fields are identified solely by their ordinal position within the struct.

Let a struct contain m fields. The struct body is encoded as follows:

1. Emit a presence bitmap with *exactly* m bits, encoded in whole bytes, least-significant bit first. The i th bit corresponds to the i th field in declaration order.
 - For a non-optional field, the bit *MUST* be set to 1.

- For an `optional<T>` field, the bit is set to 1 if the value is present and to 0 if the value is absent.
2. For each field i in declaration order:
 - If the presence bit for field i is 1, serialize the value according to the rules for its type.
 - If the presence bit for field i is 0, no bytes are emitted for that field; this case is only legal for `optional<T>` fields.

Decoders *MUST NOT* assume knowledge of the physical layout of a struct beyond the fields they explicitly recognize. After decoding the fields they understand, implementations *MUST* skip any remaining trailing bytes in the struct body as determined by the length prefix.

Trailing bytes within the struct body are reserved for fields appended in later schema versions and *MUST* be preserved during forwarding or re-encoding.

Because field ordinals define the on-wire layout, implementations *MUST NOT* reorder fields within a package version.

6.6 Optional Fields

An `optional<T>` value *MAY* be present or absent. Optional fields in structs use the presence bitmap described above. Any other type or composite type may be used as `T`.

6.7 Error Handling

A deserializer *MUST* reject any of the following:

- malformed VarUInt encodings,
- lengths that exceed the remaining buffer,
- invalid UTF-8 in `string` values,
- duplicate keys in a `map<K,V>`,
- unexpected trailing bytes after a serialized value,
- any violation of the structural constraints defined in this section.

Unless explicitly specified, deserialization errors *MUST* be treated as fatal to the enclosing RPC invocation.

6.8 Binary Layout Example

This section provides a concrete example illustrating how arf encodes structs and how length-prefixing enables safe evolution.

Consider the following initial struct definition:

```

1 struct User {
2     id    uint32;
3     name  string;
4 }
```

This struct contains two fields. A serialized instance is encoded as:

Component	Description
VarUInt(L)	Length of the struct body in bytes
Bitmap	2 bits indicating presence of fields
Field 0	id (uint32)
Field 1	name (string)

Assume an evolution where a new optional field is appended:

```

1 struct User {
2     id    uint32;
3     name  string;
4     email optional<string>;
5 }
```

The new binary layout becomes:

Component	Description
VarUInt(L')	New struct body length
Bitmap	3 bits (one for each field)
Field 0	id
Field 1	name
Field 2	email (if present)

When an older receiver (that only knows the two-field layout) decodes this value, it proceeds as follows:

- It reads `VarUInt(L')` and learns the total size of the struct body.

- It decodes only the fields it recognizes (the first two).
- It skips the remaining bytes using the length prefix.

Because all struct values are length-delimited, receivers never need to understand the internal layout of unknown fields in order to skip them safely. This guarantees forward compatibility for nested structs, arrays of structs, and arbitrarily complex type graphs.

7 RPC Protocol

arf defines a bidirectional message-oriented protocol used to transport serialized request and response values between a client and a server. This section defines the framing model, identifier space, message semantics, and the legal frame sequences for each method form defined in Section 7.6.

Unless otherwise noted, all multi-byte integer fields are encoded in little-endian byte order. All frames are self-contained and do not rely on lower-layer segmentation semantics.

7.1 Frame Format

Each message exchanged between a client and server is encoded as an arf *frame*. Frames MUST NOT be fragmented or interleaved at the byte level. The format of a frame is:

Field	Description
Magic[2]	Fixed value 0xAF 0x01 identifying arf framing.
Version[1]	Protocol version. This document specifies version 1.
MessageKind[1]	Indicates the semantic type of the frame.
Flags[1]	Reserved for future use; MUST be zero for version 1.
PackageID[4]	Identifier for the package containing the service.
ServiceID[4]	Identifier for the service.
MethodID[4]	Identifier for the method.
CorrelationID[8]	Opaque identifier linking frames belonging to the same RPC.
PayloadLength (var)	Length of the payload using VarUInt encoding.
Payload (var)	Serialized value according to Section 6.

All frames MUST begin with the 2-byte magic value. Receivers MUST treat a mismatched magic value as a framing error and close the connection.

7.1.0.1 MessageKind The `MessageKind` byte MUST be one of the following:

Value	Meaning
0x01	INVOKE: initiates an RPC call.
0x02	IN_STREAM: element of an input stream.
0x03	IN_CLOSE: signals end of input stream.
0x04	OUT_STREAM: element of an output stream.
0x05	OUT_CLOSE: signals end of output stream.
0x06	RESPONSE: unary output payload.
0x07	ERROR: terminal error.
0x08	CANCEL: client cancellation of the RPC.

7.2 Transport Assumptions

arf is designed to run over a reliable, ordered, bidirectional transport. The transport **MUST** preserve byte order and **MUST NOT** duplicate or reorder bytes. Examples of suitable transports include TCP connections, QUIC streams, and Unix domain sockets.

arf frames, as defined in Section 7.1, are logical protocol units and do not necessarily correspond to transport-level segments. Endpoints **MUST** be able to reconstruct frames from an arbitrary segmentation of the underlying byte stream.

Datagram-oriented transports (such as bare UDP) do not satisfy these requirements without additional reliability and ordering mechanisms. Such mechanisms are out of scope for this specification.

7.3 Connection Model and Multiplexing

arf is designed to support multiplexing of multiple RPCs over a single transport connection. A connection is viewed as a bidirectional stream of frames as defined in Section 7.1.

7.3.1 Correlation Identifiers

The **CorrelationID** field in each frame identifies the logical RPC to which the frame belongs. The following rules apply:

- For a given connection, the client **MUST** choose a **CorrelationID** that is not currently active when initiating a new RPC.
- An RPC becomes active when its **INVOKE** frame is sent. It remains active until the RPC completion conditions defined in Section 7.6.1 are satisfied.
- Once an RPC has completed, its **CorrelationID** **MAY** be reused for a subsequent RPC on the same connection.

Implementations **MUST** treat the receipt of frames for an unknown or inactive **CorrelationID** (that is, a **CorrelationID** for which no active RPC exists) as a protocol error.

7.3.2 Concurrent RPCs

Clients MAY initiate multiple RPCs concurrently over the same connection by sending multiple `INVOKE` frames with distinct `CorrelationID` values. Similarly, servers MAY process multiple RPCs concurrently and interleave frames for those RPCs arbitrarily.

For each individual `CorrelationID`, the following invariants MUST hold:

- Exactly one `INVOKE` frame MUST appear, and it MUST be the first frame for that `CorrelationID`.
- All subsequent frames with that `CorrelationID` MUST conform to the method form and sequencing rules defined in Section 7.6.
- Once the RPC completion conditions defined in Section 7.6.1 have been satisfied, endpoints MUST treat the `CorrelationID` as inactive and MUST NOT send further frames for that `CorrelationID`.

Frames belonging to different `CorrelationID` values MAY be arbitrarily interleaved in both directions, subject to any flow-control or prioritization policies implemented by the endpoints.

7.3.3 Ordering Guarantees

The transport MUST preserve byte order, and endpoints MUST preserve the frame order in which data is received. As a consequence:

- For a given `CorrelationID`, the sequence of frames is strictly ordered and can be processed in order.
- No ordering guarantees are provided between frames belonging to different `CorrelationID` values beyond those implied by the underlying transport.

arf does not define fairness or prioritization between concurrent RPCs. Such policies are implementation-specific and MAY be influenced by application-level considerations.

7.4 Identifiers

`PackageID`, `ServiceID`, and `MethodID` are 32-bit unsigned integers derived from the fully-qualified names of packages, services, and methods using the FNV-1a hash function. The hash function MUST be implemented exactly as specified in this section; all conforming implementations MUST produce identical identifiers for the same fully-qualified name.

The identifier derivation scheme is intentionally opaque in arf source files and MUST NOT be user-configurable. Identifiers MUST remain stable for the lifetime of a package version. Changing any identifier constitutes a breaking change requiring a new package version.

7.4.1 FNV-1a Hash Function

arf derives **PackageID**, **ServiceID**, and **MethodID** values using the FNV-1a non-cryptographic hash function operating on 32-bit unsigned integers. FNV-1a is selected for its simplicity, determinism, efficiency, and well-defined behavior across implementations.

The FNV-1a algorithm **MUST** be implemented exactly as specified in this section. All conforming implementations **MUST** produce identical hash outputs for the same input byte sequence.

The hash function operates over a sequence of octets and produces a single 32-bit unsigned integer. Arithmetic is performed using unsigned 32-bit modular arithmetic with wraparound on overflow.

FNV-1a is not a cryptographic hash function and **MUST NOT** be used for security-sensitive purposes such as message authentication, integrity verification, or identity proof. Its sole purpose in arf is deterministic identifier derivation.

The domain of the input to the hash function is restricted and well-defined. Implementations **MUST** apply namespace-specific prefixes prior to hashing to ensure that identifiers for different kinds of entities do not collide. The input to the hash function for each identifier is defined as follows:

- **PackageID** is derived from: "pkg:" || <package-fqn>
- **ServiceID** is derived from: "svc:" || <service-fqn>
- **MethodID** is derived from: "method:" || <package> "." <service> "." <method>

The fully-qualified name strings **MUST** be encoded as UTF-8 prior to hashing, without terminating null bytes or additional normalization.

Collisions **MUST** be detected at compile time. If two distinct fully-qualified names produce the same identifier within the same identifier space (package, service, or method), the compiler **MUST** reject the program. Implementations **MUST NOT** accept such collisions silently.

Implementations **MUST** treat the input byte stream as an ordered sequence of octets. No Unicode normalization, case-folding, or locale-sensitive transformation is permitted.

The normative description and reference implementation of the FNV-1a algorithm appear below.

```
FNV-1a-32(input):
```

```
Let offset_basis = 2166136261 (0x811C9DC5).
Let FNV_prime    = 16777619   (0x01000193).
```

```

Let h = offset_basis.
For each byte b in input, in order:
    h = h XOR b
    h = (h * FNV_prime) mod 2^32

Return h.

```

A C reference implementation and test vectors are provided in Appendix A and Appendix B, respectively.

7.5 Requests

An RPC request is initiated by the client sending an **INVOKE** frame. The payload of the **INVOKE** frame depends on the method form:

- If **HasInput** is true, the payload **MUST** contain the serialization of the unary input struct.
- If **HasInput** is false, the payload **MUST** be empty.
- If **HasInputStream** is true, the input stream is considered open immediately after the **INVOKE** frame. The client **MAY** send zero or more **IN_STREAM** frames, followed by exactly one **IN_CLOSE**.

A client **MUST NOT** send additional **INVOKE** frames with the same **CorrelationID**. Doing so constitutes a protocol error.

7.6 Method Forms and Allowed Frame Sequences

arf defines sixteen distinct *method forms*, derived from the presence or absence of: (1) unary input parameters, (2) unary output values, (3) an input stream, and (4) an output stream. The four-letter form label encodes the booleans **HasInput**, **HasOutput**, **HasInputStream**, and **HasOutputStream**, respectively, using **Y** for “present” and **N** for “absent”.

Table 1 specifies, for each form, whether the client and server **MAY** send the streaming frame kinds defined in Section 7.1. Frames belonging to different **CorrelationID** values **MAY** be arbitrarily interleaved on the same connection, as described in Section 7.3.

Legend:

- **HasIn** = **HasInput**
- **HasOut** = **HasOutput**
- **HasInS** = **HasInputStream**

Table 1: arf Method Forms and Permitted Streaming Frames (per `CorrelationID`)

Form	HasIn	HasOut	HasInS	HasOutS	C:IS	C:IC	S:OS	S:OC
NNNN	N	N	N	N	N	N	N	N
NNNY	N	N	N	Y	N	N	Y	Y
NNYN	N	N	Y	N	Y	Y	N	N
NNYY	N	N	Y	Y	Y	Y	Y	Y
NYNN	N	Y	N	N	N	N	N	N
NYNY	N	Y	N	Y	N	N	Y	Y
NYYN	N	Y	Y	N	Y	Y	N	N
NYYY	N	Y	Y	Y	Y	Y	Y	Y
YNNN	Y	N	N	N	N	N	N	N
YNNY	Y	N	N	Y	N	N	Y	Y
YNYN	Y	N	Y	N	Y	Y	N	N
YNYY	Y	N	Y	Y	Y	Y	Y	Y
YYNN	Y	Y	N	N	N	N	N	N
YYNY	Y	Y	N	Y	N	N	Y	Y
YYYN	Y	Y	Y	N	Y	Y	N	N
YYYY	Y	Y	Y	Y	Y	Y	Y	Y

- HasOutS = HasOutputStream
- C:IS = client MAY send `IN_STREAM`
- C:IC = client MAY send `IN_CLOSE`
- S:OS = server MAY send `OUT_STREAM`
- S:OC = server MAY send `OUT_CLOSE`

For each form, the following additional constraints apply, per `CorrelationID`:

- Exactly one `INVOKE` frame MUST be sent, and it MUST be the first frame.
- If HasInS is Y, the client MAY send zero or more `IN_STREAM` frames followed by exactly one `IN_CLOSE`. If HasInS is N, the client MUST NOT send `IN_STREAM` or `IN_CLOSE`.
- If HasOutS is Y, the server MAY send zero or more `OUT_STREAM` frames followed by exactly one `OUT_CLOSE`. If HasOutS is N, the server MUST NOT send `OUT_STREAM` or `OUT_CLOSE`.

7.6.1 RPC Completion

For a given `CorrelationID`, an RPC is considered complete when one of the following conditions holds:

- An **ERROR** frame has been received, or
- A **CANCEL** frame has been received, or
- A **RESPONSE** frame has been received and all streams declared by the method form have been closed:
 - if **HasInputStream** is **true**, an **IN_CLOSE** frame has been received; and
 - if **HasOutputStream** is **true**, an **OUT_CLOSE** frame has been received.

Once any of the above conditions is satisfied, endpoints **MUST** treat the **CorrelationID** as inactive and **MUST NOT** send further frames with that **CorrelationID**.

7.7 Interleaving of Input and Output Streams

For method forms that declare both an input stream and an output stream (i.e., **HasInputStream** = **true** and **HasOutputStream** = **true**), the client and server **MAY** send **IN_STREAM** and **OUT_STREAM** frames in any order, including arbitrarily interleaved. Neither endpoint is required to await stream activity from the peer before sending additional elements.

Stream directions are independent: the client **MAY** continue sending input elements regardless of whether the server is actively sending output elements, and vice versa.

7.8 Unary vs Streaming Semantics

7.8.1 Unary Input

If **HasInput** is **true**, the **INVOKE** payload **MUST** contain a serialized unary input struct. If **false**, the payload **MUST** be empty.

7.8.2 Input Stream

If **HasInputStream** is **true**:

- The client **MAY** send zero or more **IN_STREAM** frames.
- Each **IN_STREAM** frame **MUST** contain exactly one serialized element of the declared input stream type.
- The client **MUST** send exactly one **IN_CLOSE**.
- After **IN_CLOSE**, no further inbound stream frames are permitted.

7.8.3 Unary Output

For every RPC that completes successfully (that is, does not terminate with an **ERROR** frame), the server **MUST** send exactly one **RESPONSE** frame for the corresponding **CorrelationID**, regardless of whether the method declares no unary outputs, one or more unary outputs, an output stream, or any combination thereof.

The **RESPONSE** frame **MUST** be the first application-level frame sent by the server for that **CorrelationID**. In particular, the server **MUST NOT** send any **OUT_STREAM** or **OUT_CLOSE** frames before the **RESPONSE** frame.

If **HasOutput** is **true**, the payload of the **RESPONSE** frame **MUST** contain the serialized unary output struct or tuple. If **HasOutput** is **false**, the payload of the **RESPONSE** frame **MUST** be empty.

7.8.4 Unary Output Ordering

For all method forms in which **HasOutput** is true, the server **MUST** send the **RESPONSE** frame before transmitting any **OUT_STREAM** frames. Once **RESPONSE** has been sent, the server **MAY** interleave **OUT_STREAM** frames with incoming **IN_STREAM** frames arbitrarily.

7.8.5 Output Stream

If **HasOutputStream** is **true**, the server **MAY** produce an output stream *after* sending the **RESPONSE** frame.

In that case, the following rules apply for the given **CorrelationID**:

- The server **MAY** send zero or more **OUT_STREAM** frames.
- Each **OUT_STREAM** frame **MUST** contain exactly one serialized element of the declared output stream type.
- The server **MUST** send exactly one **OUT_CLOSE** frame to indicate the end of the output stream.
- The server **MUST NOT** send any **OUT_STREAM** or **OUT_CLOSE** frames before the **RESPONSE** frame.
- After sending **OUT_CLOSE**, the server **MUST NOT** send any further **OUT_STREAM** or **OUT_CLOSE** frames for that **CorrelationID**.

Thus, for methods with **HasOutputStream** set to **true**, the canonical successful server-side sequence (ignoring input frames) is:

```

RESPONSE
OUT_STREAM*    // zero or more times
OUT_CLOSE

```

7.9 Per-form Examples

This subsection provides illustrative but non-normative examples of legal frame sequences for selected method forms. Let `CID` be a `CorrelationID`.

7.9.1 Form: NNNN (no input, no output, no streams)

```
INVOKE(CID)
RESPONSE(CID)
```

7.9.2 Form: NNNY (server-streaming only)

```
INVOKE(CID)
RESPONSE(CID)
OUT_STREAM(CID)
OUT_STREAM(CID)
OUT_CLOSE(CID)
```

7.9.3 Form: YNYN (unary input, input stream, no unary output)

```
INVOKE(CID, unary-input)
RESPONSE(CID) // empty response
IN_STREAM(CID, T)
IN_STREAM(CID, T)
IN_CLOSE(CID)
```

7.9.4 Form: YYYY (full bidirectional streaming + unary input/output)

```
INVOKE(CID, unary-input)
IN_STREAM(CID, T)
```

```
IN_STREAM(CID, T)
IN_CLOSE(CID)

RESPONSE(CID, unary-output)

OUT_STREAM(CID, U)
OUT_STREAM(CID, U)
OUT_CLOSE(CID)
```

7.10 Cancellation and Error Semantics

A client MAY abort an ongoing RPC using a **CANCEL** frame with the same **CorrelationID** as the active call. Upon receiving **CANCEL**, the server:

- MUST cease sending further frames for that **CorrelationID**,
- MAY release any associated resources immediately,
- MAY send an **ERROR** frame if additional diagnostic information is available.

A server MAY emit an **ERROR** frame at any time to signal an irrecoverable condition. Upon receiving **ERROR**, the client MUST treat the RPC as terminated and MUST NOT send additional frames for that **CorrelationID**.

7.11 Backpressure

arf does not define a mandatory flow-control mechanism. Implementations SHOULD apply backpressure using transport-level or application-level means. Implementations MUST remain robust in the presence of senders that produce data more rapidly than the receiver can process.

Extensions to provide explicit credit-based flow control MAY be defined in future versions of this specification.

8 Evolution and Compatibility

arf is designed to permit incremental schema evolution without requiring explicit field identifiers in source files. Compatibility is achieved by combining stable field ordinals with length-prefixed struct encoding, allowing new fields to be added while preserving the ability of older implementations to safely skip unknown data.

The rules in this section govern how types may change across package versions while maintaining compatibility at the binary level.

8.1 Adding Fields

New fields MAY be appended to the end of an existing struct. Because field ordinal indices are defined by declaration order, appending a field assigns it the next available ordinal. Existing ordinals MUST NOT change.

Struct values are length-prefixed as defined in Section 6. As a result, receivers MUST skip any trailing bytes in the struct body beyond the fields they recognize. This allows new fields to be introduced in a backward-compatible and forward-compatible manner.

Senders MAY include newly added fields when communicating with older receivers; older receivers MUST ignore trailing bytes safely. Likewise, senders MAY omit fields not known to the peer.

New fields SHOULD be declared as `optional<T>` unless the semantics require mandatory presence. Using `optional<T>` maximizes flexibility and minimizes compatibility risk.

8.2 Removing Fields

Fields MUST NOT be removed from a struct in a way that changes the ordinal position of any remaining field. Instead, fields that are no longer used SHOULD be marked with the `@deprecated` attribute and retained with their original ordinal position.

Senders SHOULD omit deprecated fields when serializing. Receivers MUST accept and ignore deprecated fields if they appear.

If a field must be fully removed (such that its ordinal becomes invalid), this MUST be done only by introducing a new package version (see Section 8.4). No cross-package compatibility is required.

8.3 Changing Types

A field's type MUST NOT change within the same package version. Any modification to a field's type—including widening (e.g., `int32` to `int64`), narrowing, or structural change—constitutes a backward-incompatible change.

Type changes MUST be performed only by introducing a new package version.

Changing a field from a concrete type to `optional<T>` within the same package version is permitted only if:

- the wire encoding is identical when the value is present, and
- receivers can safely interpret the absence of the value as “not present”.

Changing a field from `optional<T>` to a non-optional type is not compatible and MUST NOT be performed within the same package version.

8.4 Versioning Through Packages

arf relies on the `package` namespace as the primary unit of versioning. A change that violates any of the compatibility rules in this section **MUST** be introduced under a new package name.

Package names **SHOULD** include a stability level (e.g., `v1beta1`, `v1`, `v2`) to provide clear evolution points. Implementations **MAY** simultaneously support multiple package versions.

No cross-package binary compatibility is required or implied. Changes within a package version **MUST** adhere to the evolution rules above.

9 ABNF Grammar

This section provides the normative syntax of the arf source language, expressed using Augmented Backus–Naur Form (ABNF) as defined in RFC 5234 and RFC 7405. ABNF rules specify the lexical and syntactic structure of valid arf source files. The grammar below is complete for this specification.

```
arf-source      = *c-nl source-root *c-nl

; COMMON DEFINITIONS

CRLF           = (CR LF) / LF

ident          = ALPHA *(ALPHA / DIGIT / "_")

semi           = ";" *WSP *c-nl

c-nl           = comment / CRLF ; comment line or a bare newline

comment        = "#" *(WSP / VCHAR) CRLF

snake-ident    = (%x61-7A / "_") *(%x61-7A / "_" / DIGIT)

screaming-snake-ident =
    (%x41-5A / "_") *(%x41-5A / "_" / DIGIT)

camel-ident    = %x41-5A *(ALPHA / DIGIT)

; TYPES

type           = composite-type / plain-type

plain-type     = snake-ident / camel-ident

composite-type = optional-type / array-type / map-type
```

```

optional-type      = "optional<" *WSP type *WSP ">"

array-type         = "array<" *WSP type *WSP ">"

map-type           = "map<" *WSP map-key-type *WSP "," *WSP type *WSP ">"

map-key-type       = plain-type

stream             = "stream" 1*WSP camel-ident

; SOURCE ROOT

source-root        = package *c-nl imports *c-nl definitions *c-nl

definitions        = *( *c-nl definition *c-nl )

definition         = struct / enum / service

; PACKAGE

package            = "package" 1*WSP package-name *WSP semi

package-name       = snake-ident *( "." snake-ident )

; IMPORTS

imports            = *c-nl *( "import" 1*WSP import-path [import-alias] semi )

import-path        = *WSP DQUOTE import-components DQUOTE *WSP

import-components  = 1*( ALPHA / DIGIT / "-" / "_" / "/" / "." )

import-alias       = 1*WSP "as" 1*WSP snake-ident

; ANNOTATIONS

annotations        = *( annotation )

annotation         = "@" snake-ident *WSP
                    [ "(" *annotation-params ")" ]
                    *WSP *c-nl

annotation-params  = annotation-param *( "," annotation-param )

annotation-param    = DQUOTE *VCHAR DQUOTE

```



```

; STRUCT

struct          = annotations struct-def struct-body *c-nl

struct-def      = "struct" 1*WSP camel-ident *WSP

struct-body     = "{" *struct-elements "}" *WSP *c-nl

struct-elements = *(c-nl / struct-field / struct)

struct-field    = annotations *CRLF *WSP
                  snake-ident 1*WSP type semi

; ENUM

enum           = annotations enum-def enum-body *c-nl

enum-def       = "enum" 1*WSP camel-ident *WSP

enum-body      = "{" *enum-elements "}" *WSP *c-nl

enum-elements  = *(c-nl / WSP / enum-value)

enum-value     = annotations *CRLF *WSP
                  screaming-snake-ident *WSP
                  "=" *WSP 1*DIGIT semi

; SERVICE

service        = annotations service-def service-body *c-nl

service-def     = "service" 1*WSP camel-ident *WSP

service-body    = "{" *service-elements "}" *WSP *c-nl

service-elements = *(c-nl / WSP / service-rpc)

service-rpc     = service-rpc-prelude
                  service-rpc-params
                  [ service-rpc-returns ]
                  semi

service-rpc-prelude =
                  annotations *CRLF *WSP camel-ident *WSP

service-rpc-params =
                  "(" *service-rpc-input ")" *WSP

```

```

service-rpc-input =
    stream /
    (snake-ident 1*WSP camel-ident
     *(*WSP "," *WSP snake-ident 1*WSP camel-ident)
     [*WSP "," stream])

service-rpc-returns =
    "->" (
        (*WSP camel-ident *WSP) /
        ("(" *WSP camel-ident *WSP
         *(*WSP "," *WSP camel-ident *WSP)
         [*WSP stream] *WSP ")") /
        (*WSP stream *WSP)
    )

```

10 Security Considerations

arf describes a serialization format and an RPC protocol but does not mandate any particular transport security mechanism. When deployed over an untrusted network, implementations **MUST** provide confidentiality, integrity, and endpoint authentication using a transport such as TLS [RFC8446], QUIC [RFC9000] with TLS 1.3 [RFC9001], or an equivalent secure channel.

Endpoints **MUST** validate that incoming frames conform to the arf framing rules, including length fields, frame ordering constraints, and stream termination semantics. Malformed or truncated frames **MUST** result in termination of the associated **CorrelationID** and **SHOULD** result in connection closure.

Implementations **MUST** impose limits on:

- maximum frame length,
- maximum number of concurrent **CorrelationID** streams,
- maximum nesting depth for type decoding,
- maximum aggregate memory per connection.

These limits prevent resource exhaustion attacks.

Optional fields and variable-length structures (arrays, maps, byte sequences) **MUST** be validated before allocation. Implementations **MUST** reject encodings that imply unreasonable memory commitments.

Method names, package names, and type names are not security boundaries. Applications **MAY** impose authorization checks based on method identity, but such checks are outside the scope of this specification.

Applications that embed sensitive data in arf messages **MUST** consider application-level encryption or tokenization if end-to-end confidentiality is required beyond the transport layer.

11 IANA Considerations

This document makes no requests of the IANA.

Future versions of arf MAY define frame-type registries or well-known package namespaces. Such extensions MUST define their own IANA interactions as appropriate.

12 References

- [RFC2119] Scott Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. RFC 2119. Internet Engineering Task Force, 1997-03. DOI: 10.17487/RFC2119. URL: <https://www.rfc-editor.org/rfc/rfc2119>.
- [RFC9000] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. Internet Engineering Task Force, 2021-05. DOI: 10.17487/RFC9000. URL: <https://www.rfc-editor.org/rfc/rfc9000>.
- [RFC8174] Barry Leiba. *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words*. RFC 8174. Internet Engineering Task Force, 2017-05. DOI: 10.17487/RFC8174. URL: <https://www.rfc-editor.org/rfc/rfc8174>.
- [RFC8446] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Internet Engineering Task Force, 2018-08. DOI: 10.17487/RFC8446. URL: <https://www.rfc-editor.org/rfc/rfc8446>.
- [RFC9001] Martin Thomson and Sean Turner. *Using TLS to Secure QUIC*. RFC 9001. Internet Engineering Task Force, 2021-05. DOI: 10.17487/RFC9001. URL: <https://www.rfc-editor.org/rfc/rfc9001>.

13 Acknowledgements

The author thanks the contributors to early discussions on arf’s design, particularly those who provided feedback on streaming semantics, package versioning, and binary framing constraints.

The structure and terminology of this document were influenced by the style of the IETF and prior work on schema-based RPC systems, including Protocol Buffers, Cap’n Proto, and Thrift.

14 Appendices

A FNV-1a-32 Reference Implementation

```

1 #define ARF_FNV1A32_OFFSET 0x811C9DC5u
2 #define ARF_FNV1A32_PRIME 0x01000193u
3
4 /* Compute FNV-1a over an arbitrary byte sequence. */
5 uint32_t arf_fnv1a32(const uint8_t *data, size_t len) {
6     uint32_t h = ARF_FNV1A32_OFFSET;
7
8     for (size_t i = 0; i < len; i++) {
9         h ^= (uint32_t)data[i];
10        h *= ARF_FNV1A32_PRIME; /* 32-bit unsigned wraparound
11                                */
12    }
13    return h;
14 }
15
16 /* Convenience helper for NUL-terminated UTF-8 strings. */
17 uint32_t arf_fnv1a32_cstr(const char *s) {
18     const uint8_t *p = (const uint8_t *)s;
19     uint32_t h = ARF_FNV1A32_OFFSET;
20
21     while (*p) {
22         h ^= (uint32_t)*p++;
23         h *= ARF_FNV1A32_PRIME;
24     }
25
26     return h;
27 }

```

B FNV-1a-32 Test Vectors (Informative)

This annex provides test vectors for the 32-bit FNV-1a hash function used by arf. All strings are encoded as UTF-8 without a terminating NUL byte unless otherwise noted.

Input string	FNV-1a-32 output
(empty string)	0x811C9DC5
"a"	0xE40C292C
"b"	0xE70C2DE5
"foobar"	0xBF9CF968
"pkg:v1beta1.common"	0xF746E480
"svc:v1beta1.common.TimestampService"	0xEAA88025
"method:v1beta1.common.TimestampService.GetTimestamp"	0x01015F42

Implementations SHOULD verify their FNV-1a-32 implementation against these test vectors before using it to derive `PackageID`, `ServiceID`, or `MethodID` values.

C C Reference Implementation for ZigZag Encoding

The following C code implements ZigZag transformation for 32-bit and 64-bit signed integers.

```

1  /* Encode signed 32-bit integer into ZigZag form. */
2  static inline uint32_t arf_zigzag32_encode(int32_t n) {
3      return (uint32_t)((((uint32_t)n << 1) ^ (uint32_t)(n >> 31)
4          );
5  }
6  /* Decode ZigZag-encoded 32-bit integer. */
7  static inline int32_t arf_zigzag32_decode(uint32_t z) {
8      return (int32_t)((z >> 1) ^ (uint32_t)-(int32_t)(z & 1u));
9  }
10
11 /* Encode signed 64-bit integer into ZigZag form. */
12 static inline uint64_t arf_zigzag64_encode(int64_t n) {
13     return (uint64_t)((((uint64_t)n << 1) ^ (uint64_t)(n >> 63)
14         );
15 }
16 /* Decode ZigZag-encoded 64-bit integer. */
17 static inline int64_t arf_zigzag64_decode(uint64_t z) {
18     return (int64_t)((z >> 1) ^ (uint64_t)-(int64_t)(z & 1u));
19 }

```

D Test Vectors for Signed Integer Encoding

The following test vectors are normative. Implementations *MUST* produce exactly the given ZigZag and VarUInt results.

D.1 Common Values

Value	ZigZag(n)	Decimal	VarUInt Encoding
0	0x00	0	00
-1	0x01	1	01
1	0x02	2	02
-2	0x03	3	03
2	0x04	4	04
63	0x7E	126	7E
-64	0x7F	127	7F
64	0x80	128	80 01
-65	0x81	129	81 01
300	0x0258	600	D8 04
-300	0x0257	599	D7 04

D.2 Boundary Values

D.2.1 int8

Value	ZigZag(n)	VarUInt Bytes
-128	0xFF	FF 01
127	0xFE	FE 01

D.2.2 int16

Value	ZigZag(n)	VarUInt Bytes
-32768	0xFFFF	FF FF 03
32767	0xFFFE	FE FF 03

D.2.3 int32

Value	ZigZag(n)	VarUInt Bytes
-2147483648	0xFFFFFFFF	FF FF FF FF 0F
2147483647	0xFFFFFFFFE	FE FF FF FF 0F

D.2.4 int64

	Value	ZigZag(n)	VarUInt Bytes
INT64_MIN	0xFFFFFFFFFFFFFFFF	FF FF FF FF FF FF FF FF	01
INT64_MAX	0xFFFFFFFFFFFFFFFFE	FE FF FF FF FF FF FF FF	01

E Implementation Notes (Informative)

This appendix provides non-normative guidance for implementers of arf encoders, decoders, and RPC runtimes. The recommendations in this section are intended to improve robustness, interoperability, and operational safety, but they do not introduce additional protocol requirements beyond the normative text in the main body of this specification.

E.1 Recommended Limits

Although the core specification does not mandate concrete numeric limits, practical implementations SHOULD enforce the following upper bounds:

- **Maximum frame size:** Implementations SHOULD bound the size of a single arf frame (including header and payload). A value on the order of 1–16 MiB is typical for RPC-style workloads, but deployments MAY choose tighter or looser limits based on use case.
- **Maximum struct body size:** Independently from the frame limit, implementations SHOULD bound the decoded struct body length (the `VarUInt(L)` prefix for a struct). This prevents a malicious peer from embedding an excessively large struct within an otherwise reasonable frame.
- **Maximum nesting depth:** Decoders SHOULD enforce a maximum nesting depth for composite types (e.g., structs containing arrays of structs containing maps, and so on). A depth limit on the order of 32–64 is generally sufficient to prevent stack exhaustion in recursive decoders.
- **Maximum collection lengths:** Implementations SHOULD impose limits on:
 - maximum array length,
 - maximum map entry count,
 - maximum string length (in bytes),
 - maximum `bytes` length.

These limits SHOULD be enforced before allocating memory for the collection payload.

- **Maximum concurrent RPCs:** Servers SHOULD bound the number of simultaneously active `CorrelationID` values per connection, and MAY apply per-principal or per-IP concurrency limits at a higher layer.

Implementations SHOULD treat the violation of such local limits as a fatal error for the affected RPC, and MAY close the underlying connection if abuse is suspected.

E.2 VarUInt and Signed Integer Decoding

Implementations SHOULD adopt the following practices when decoding `VarUInt` and ZigZag-encoded integers:

- **Byte limit:** For 64-bit values, at most ten bytes are needed to represent any valid `VarUInt`. Decoders SHOULD reject encodings that use more than ten bytes, even if the resulting numeric value would still fit in a 64-bit unsigned integer.
- **Early overflow detection:** When reconstructing the integer value, decoders SHOULD detect overflow incrementally (for example, before shifting or adding each new 7-bit chunk) rather than relying on native wraparound.
- **Width checking:** After applying ZigZag decoding, implementations MUST verify that the result falls within the range of the declared signed type (as specified in Section 6.3). Implementers are encouraged to centralize these checks in shared helpers to avoid inconsistencies across call sites.

For encoder implementations, it is RECOMMENDED to reuse the same `VarUInt` and ZigZag routines across all integer types to ensure uniform behavior and simplify testing.

E.3 Framing and Buffering Strategies

Because arf frames are carried over a reliable byte stream, endpoints MUST reconstruct frames from arbitrary segment boundaries. In practice, many implementations adopt one of the following patterns:

- Maintain a per-connection read buffer, append bytes as they are received, and attempt to parse complete frames in a loop until no further complete frame is available.
- Use a small state machine that first parses the fixed header, then decodes `PayloadLength` using `VarUInt`, and finally waits until the indicated number of payload bytes are available before dispatching the frame.

Implementations SHOULD validate the `Magic` and `Version` fields before allocating large buffers for the payload. If either field is invalid, the connection SHOULD be closed immediately.

E.4 Stream and RPC Lifecycle Management

Implementers are encouraged to model each active RPC as an explicit state machine keyed by `CorrelationID`. Such a state machine typically tracks:

- whether `INVOKE` has been seen,
- whether `RESPONSE` has been sent or received,
- whether the input stream (if any) is open or closed,
- whether the output stream (if any) is open or closed,
- whether an `ERROR` or `CANCEL` has terminated the RPC.

Once the RPC completion conditions in Section 7.6.1 are satisfied, the corresponding state **MUST** be discarded and the `CorrelationID` returned to the pool of available identifiers.

To avoid resource leaks, implementations **SHOULD** apply inactivity timeouts to RPCs that do not progress (for example, an input stream that never sends `IN_CLOSE`, or a server that never produces `RESPONSE`). Exceeding such a timeout **MAY** result in an `ERROR` frame and/or connection closure.

E.5 Timestamps and Clock Handling

arf `timestamp` values represent milliseconds since the UTC Unix Epoch. The following practices are **RECOMMENDED**:

- Timestamps **SHOULD** be generated based on a monotonic, reasonably accurate system clock synchronized via a time protocol such as NTP or equivalent.
- When converting between local time and UTC, implementations **SHOULD** use a time-zone aware library rather than hard-coding offsets, in order to correctly handle daylight saving changes and historical time-zone adjustments.
- When a platform cannot represent millisecond precision, it **SHOULD** truncate fractional milliseconds toward zero when encoding, as described in the core specification.

When comparing timestamps originating from different systems, applications **SHOULD** be tolerant to moderate clock skew and **SHOULD** avoid treating small discrepancies as protocol errors.

E.6 Identifier Caching

Because FNV-1a identifier derivation is deterministic and depends only on fully-qualified names and fixed prefixes, implementations **MAY** cache the resulting **PackageID**, **ServiceID**, and **MethodID** values to avoid recomputing them at runtime.

Typical approaches include:

- precomputing identifiers at code-generation time and emitting them as constants in generated stubs, or
- computing identifiers once at process start and storing them in an immutable lookup table keyed by fully-qualified name.

Regardless of caching strategy, compilers and build tooling **MUST** still detect collisions during schema compilation, as specified in Section 7.4.

E.7 Error Mapping and Diagnostics

arf does not define a structured error payload format. Implementations typically wrap **ERROR** frames with an application-specific structure that includes:

- a machine-readable error code,
- a human-readable message,
- optional metadata such as retry hints or correlation tokens.

It is **RECOMMENDED** that runtimes map transport- or decoding-level failures (such as malformed frames, invalid VarUInt encodings, or type width violations) to **ERROR** frames only when the connection remains otherwise usable. If the integrity of the framing layer is in doubt (for example, due to a bad magic number or a severely malformed header), the implementation **SHOULD** close the connection instead of attempting to recover.