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

Contents

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 7.8.2 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 an explicit presence marker in the serialized representation (see Section 6.7).

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:

```
package v1beta1.myservice;
```

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

Example:

```
v1beta1.mypackage.OtherName
```

5.2 Imports

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

```
import "../../common/v1beta1/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 (such as integers, booleans, floating-point values, strings, timestamps, and byte sequences) and composite types (such as `array<...>`, `map<...>`, and `optional<...>`) MUST NOT appear directly as unary method parameters, unary method results, or stream element types. Only user-defined `struct` and `enum` types are permitted in RPC method signatures (see Section 5.7.2).

- `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 a signed integer count of milliseconds elapsed since the Unix Epoch (`1970-01-01T00:00:00Z`). Values prior to the Unix Epoch are represented as negative integers and are encoded using the signed integer encoding rules defined in Section 6.3.
- `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`. Only the following map key types (for `K`) are permitted:
  - any unsigned or signed integer type, and
  - any `enum` type.

  All other types are forbidden as map keys.
- `optional<T>`: a value of type `T` that MAY be present or absent.

User-defined `struct` and `enum` 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.

Enums are encoded as unsigned integer discriminants in the range $\{x \in \mathbb{Z} \mid 0 \leq x \leq 65535\}$, serialized using the `VarUInt` scheme. The enum declaration and encoding rules are defined in Section 5.4.

5.4 Enum Declarations

Enums define a finite set of named constants, each associated with an explicit unsigned integer discriminant. Enum values are commonly used as symbolic constants, status codes, and protocol signals.

An enum declaration has the following form:

```
enum HTTPStatus {
    OK          = 200;
    NOT_FOUND   = 0x194;
    TEAPOT      = 0x1A2;
}
```

Each enumeration value MUST specify an explicit, non-negative integer discriminant using the `=` syntax. The following rules apply:

- Discriminant values MAY be reused within the same enum. If multiple enumeration members share the same discriminant, they are semantically equivalent at the wire level and MUST be treated as aliases.
- Discriminants MUST be elements of the set $\{x \in \mathbb{Z} \mid 0 \leq x \leq 65535\}$.
- Decimal and hexadecimal literals MAY be used.
- The compiler MUST reject out-of-range discriminants.

The textual ordering of enumeration values does *not* affect their numeric representation on the wire; only the explicitly assigned discriminant value is serialized.

Enum names and value identifiers are subject to the same scoping and naming rules as other top-level declarations within a package.

5.4.1 Encoding

Enumeration values are serialized as unsigned integers using `VarUInt` encoding (Section 6.2). The encoded numeric value is the declared discriminant of the enumeration case.

Decoders MUST accept any enum value whose discriminant does not match a known case in the enum definition and MUST preserve its numeric discriminant value. Applications that require closed-set semantics SHOULD explicitly validate that a received discriminant is one of the known cases and reject or handle unknown values according to application policy.

This loose validation is intentional to allow enum definitions to evolve without breaking older receivers, while still enabling applications to enforce stricter semantics when required. In any case, discriminants MUST NOT exceed 65535 prior to `VarUInt` encoding.

5.4.2 Use as Map Keys

Enums MAY be used as keys in `map<K,V>` types. When used as a key, an enum behaves identically to its underlying unsigned integer discriminant.

Equality for enum keys is determined solely by numeric equality of the discriminant value.

5.4.3 Evolution Rules

Enum declarations MAY be extended by introducing new values with fresh discriminants.

Existing discriminants MUST NOT be changed. Removing or reassigning a discriminant constitutes a breaking change and MUST be performed only by introducing a new package version.

Compilers MAY warn if an enum change breaks monotonicity or introduces large gaps, but such gaps are not semantically invalid.

5.5 Struct Declarations

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

```
struct TeamNameRequest {
    name string;
}
```

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.

5.6 Annotations

arf supports lightweight annotations that attach metadata to declarations without affecting on-the-wire encoding. An annotation is written as an at-sign followed by an identifier, optionally with a parenthesized list of string arguments:

```
@deprecated
@deprecated("use NewType instead")
@foo
@foo()
@bar("a", "b")
```

Annotation arguments, if present, MUST be string literals. Whitespace around parentheses and commas is permitted. An empty argument list (e.g., `@foo()`) and

an omitted argument list (e.g., `@foo`) are syntactically distinct but MUST be treated equivalently by the compiler.

Annotations MAY appear immediately before any of the following:

- top-level `struct` declarations;
- top-level `enum` declarations;
- `service` declarations;
- individual `service` methods;
- struct fields;
- enumeration members.

Multiple annotations MAY be applied to the same declaration by listing them on separate lines or on the same line. Annotations MUST NOT appear in any other position.

Annotations are metadata only. All annotations are ignored by the binary encoding rules defined in Section 6 unless explicitly stated otherwise in this specification.

This specification reserves the annotation name `@deprecated`. Other annotation names MAY be defined by implementations or tooling but MUST NOT alter the binary serialization or protocol semantics defined by this specification.

The `@deprecated` annotation indicates that a declaration is obsolete and SHOULD NOT be used by new code:

- For struct fields, `@deprecated` indicates that the field remains part of the wire format (and retains its ordinal position) but SHOULD be omitted by new senders when possible. Receivers MUST continue to accept such fields if they appear.
- For types, services, and methods, `@deprecated` indicates that the declaration is retained for compatibility but SHOULD NOT be referenced by new implementations.

If arguments are provided to `@deprecated`, their contents are implementation-defined but SHOULD be treated as human-readable diagnostic text (for example, a deprecation reason or replacement hint). Compilers SHOULD emit equivalent deprecation annotations to any generated code, depending on the target language's capabilities.

Compilers and tooling SHOULD emit warnings when referencing `@deprecated` declarations. The presence or absence of `@deprecated` MUST NOT change the encoding of any value on the wire, nor affect identifier derivation.

5.7 Service Declarations

Services groups RPC methods:

```
service NameService {
    check_name(req CheckNameRequest) -> common.CheckNameResponse;
}
```

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.

5.7.1 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:

```
service NameService {
    check_name(req CheckNameRequest) -> common.CheckNameResponse;
}

service NameService {
    check_slug(req CheckSlugRequest) -> common.CheckSlugResponse;
}
```

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.

5.7.2 Methods

An arf method signature consists of zero or more unary input parameters, at most one input stream, and either zero or more unary output parameters or 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 logical method forms, but method signatures that declare unary output values and an output stream are not valid, as such methods could yield fragile APIs, and potentially impact the ergonomy of generated APIs. This makes any method whose form has **HasOutput** = `true` and **HasOutputStream** = `true` (i.e., forms NYNY, NYYY, YYNY, and YYYY) invalid, and MUST be rejected by compilers.

The remaining twelve forms are considered *legal method forms*. Each legal 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 ensures that all data transmitted through RPC methods is carried by explicitly named message types, rather than anonymous primitive or composite types, and restates the restriction introduced in Section 5.3.

For the purposes of on-the-wire representation, the ordered list of unary input parameters of a method is treated as a *unary input tuple*, and the ordered list of unary output values as a *unary output tuple*. A tuple is not a first-class type in arf; it is a serialization construct that encodes "zero or more ordered values" as a single payload. The encoding of tuples is defined in Section 6.6.

5.7.3 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.

Methods declaring an output stream MUST NOT declare any unary output values. Its output side consists solely of the output stream. In other words, method signatures in which **HasOutput** and **HasOutputStream** are both `true` are syntactically invalid in arf and MUST be rejected by compilers, as defined in Section 5.7.2.

Examples:

```
1 NNYN(stream T);
2 NYNN() -> O;
3 YNNY(i I) -> stream U;
4 YNYY(i I, stream T) -> 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.

5.8 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 5.5.

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 big-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 → ZigZag → 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 wrapped in a `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 big-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**: represents an instant on the UTC time-line, encoded as a signed 64-bit integer count of milliseconds elapsed since the Unix Epoch (`1970-01-01T00:00:00Z`). Values prior to the Unix Epoch are represented as negative integers and are encoded using the signed integer encoding rules defined in Section 6.3.

  Timestamp values represent absolute instants and do not carry timezone or offset information. 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 5.7.2.

Key equality is defined by equality of the underlying key value. For integer keys, numeric equality applies. For enumeration keys, equality is determined by numeric equality of their discriminants.

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

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. The preserved order has no semantic meaning and MUST NOT be relied upon by applications.

Keys MUST be unique according to the equality semantics of type `K`. Encoders MUST NOT emit duplicate keys and 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 length indicator that defines the total size of the struct body in bytes, allowing decoders to skip unknown types 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 serialized field values in declaration order.

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 the concatenation, in declaration order, of the encodings for each field:

1. For a field whose declared type is `optional<T>`:
   - Emit a single *presence byte*. The value `0x00` denotes "absent" and the value `0x01` denotes "present". No other values are permitted.
   - If the presence byte is `0x01`, emit the serialized representation of `T`, encoded according to the rules for type `T`.
   - If the presence byte is `0x00`, no additional bytes are emitted for that field.
2. For a field whose declared type is not `optional<...>`: emit the serialized representation of the field value according to the rules for its type. Encoders MUST NOT omit fields; decoders MUST treat truncation before the end of a required field as an error.

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.

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

6.6 Unary Tuples

Unary tuples are serialization-only aggregates used to carry the ordered list of unary input parameters or unary output values of a method as a single payload. Tuples are not declared in arf sources; they are derived mechanically from the method signature.

Let a unary tuple contain n values $v_0, v_1, \ldots, v_{n-1}$, each with a declared type T_i (where each T_i is a `struct` or `enum`, as required by Section 5.7.2). The tuple is encoded as:

1. `VarUInt(L)`, where L is the total number of bytes occupied by the concatenation of the serialized values v_i; and
2. the serialized representations of $v_0, v_1, \ldots, v_{n-1}$ in declaration order, each encoded according to the rules for its type.

Thus, the on-wire layout of a tuple is:

$$\text{VarUInt}(L) \parallel \text{encode}(v_0) \parallel \text{encode}(v_1) \parallel \ \ldots \ \parallel \text{encode}(v_{n-1})$$

A tuple MAY be empty ($n = 0$). An empty tuple is encoded as `VarUInt(0)` followed by zero bytes.

Because each tuple is length-delimited by its leading `VarUInt(L)`, decoders can skip unknown trailing values if a method signature is extended by appending unary inputs or unary outputs in a later method version, provided both sides agree on the method form and framing semantics.

6.7 Optional Fields

An `optional<T>` value MAY be present or absent. The encoding of `optional<T>` is uniform regardless of where it appears (struct field, array element, map value, or nested inside other composite types).

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

- a single presence byte, where `0x00` denotes "absent" and `0x01` denotes "present"; and
- if the presence byte is `0x01`, the serialized representation of `T`, encoded according to the rules for type `T`.

If the presence byte is `0x00`, no additional bytes are emitted for that value.

The type parameter `T` MUST be a valid arf type, including primitive types, composite types (such as `array<...>` and `map<...>`), structs, enums, or even another `optional<U>`.

When `optional<T>` is used as a struct field type, its encoding follows the same rules as above and contributes to the struct body in declaration order.

6.8 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>`,
- trailing bytes beyond a declared length prefix for any length-delimited 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.

6.9 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:

```
struct User {
    id   uint32;
    name string;
}
```

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

Component	Description
VarUInt(L)	Length of the struct body in bytes
Field 0	`id` (uint32)
Field 1 length	VarUInt indicating the length of Field 1
Field 1 value	`name` (string)

Assume an evolution where a new optional field is appended:

```
struct User {
    id    uint32;
    name  string;
    email optional<string>;
}
```

The new binary layout becomes:

Component	Description
VarUInt(L')	New struct body length
Field 0	`id` (uint32)
Field 1 length	VarUInt indicating the length of Field 1
Field 1 value	`name` (string)
Field 2 tag	Presence byte for `email` (`0x00` or `0x01`)
Field 2 value	`string` payload if present (`email`)

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 treats the remaining bytes in the struct body as unknown data and skips them 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.

All frames are self-contained and do not rely on lower-layer segmentation semantics.

Unless otherwise noted, all multi-byte integer fields not being governed by VarUInt or Zig-Zag encoding are encoded in big-endian byte order. This also applies to float values.

7.1 Frame Format

Each message exchanged between a client and server is encoded as an arf *frame*. Frames are logical protocol units: bytes belonging to two different frames MUST NOT be interleaved on the wire. A single frame MAY be split across multiple transport segments, but the concatenation of bytes delivered by the transport for that connection MUST form a well-defined sequence of complete frames. 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.
CorrelationID[8]	Opaque identifier linking frames belonging to the same RPC.
PayloadLength (var)	Length of the payload using VarUInt encoding.
Payload (var)	Frame payload. Payload layout is defined in the frame-type subsections below.

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.

7.1.1 MessageKind

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

Value	Meaning
0x01	`INVOKE`: initiates an RPC call.
0x02	`CONTINUE`: server acknowledgment of `INVOKE`.
0x03	`IN_STREAM`: element of an input stream.
0x04	`IN_CLOSE`: signals end of input stream.
0x05	`OUT_STREAM`: element of an output stream.
0x06	`OUT_CLOSE`: signals end of output stream.
0x07	`RESPONSE`: unary output payload and RPC termination.
0x08	`ERROR`: terminal error (Section 7.10.1).
0x09	`CANCEL`: client cancellation of the RPC.
0x0A	`CANCELLED`: cancellation acknowledgement (Section 7.10.2)

The semantics and payloads of each frame kind are defined in Section 7.2 and subsequent subsections. In particular:

- `CANCEL` frames MUST have a payload length of zero. The `PayloadLength` field MUST encode the value 0 using `VarUInt`, and no payload bytes MUST follow. Receivers MUST treat any `CANCEL` frame with a non-zero payload length as a protocol error.

7.1.2 INVOKE frame

`INVOKE` frames serve to indicate the intention of a client to start an RPC call. It carries the Package, Service and Method identifiers, and when **HasInput** is `true`, the unary input tuple required by the method. When **HasInput** is `false`, the `INVOKE` frame must contain an empty unary input tuple; meaning that only a `VarUInt(0)` is present.

Clients MAY assume the method has begun processing after a `CONTINUE` frame is received in response to the `INVOKE` frame. Otherwise, servers MUST return an `ERROR` frame indicating that the RPC binding failed and no action was taken.

The `INVOKE` frame is composed of the following items, in order:

- `PackageID`: the identifier of the package where the RPC method resides encoded as an unsigned 32-bit integer.
- `ServiceID`: the identifier of the service providing the RPC method encoded as an unsigned 32-bit integer.
- `MethodID`: the identifier of the method being called on the provided package and service encoded as an unsigned 32-bit integer.
- `Payload`: when **HasInput** is `true`, the unary input tuple containing all inputs required by the RPC method, as described in Section 6.6; when **HasInput** is `false`, no tuple is encoded and the input tuple must be omitted by including a single `VarUInt(0)` value, while other fields MUST be present.

7.1.3 CONTINUE Frame

The `CONTINUE` frame acknowledges that the server has successfully validated the `INVOKE` frame and that the RPC has been bound to the specified `PackageID`, `ServiceID`, and `MethodID`. After sending `CONTINUE`, the server commits to executing the RPC unless a subsequent `CANCEL` frame terminates it.

A server MUST send exactly one `CONTINUE` frame for each valid `INVOKE` it accepts. If the `INVOKE` cannot be bound (unknown package, service, or method), the server MUST send an `ERROR` frame instead of `CONTINUE`.

The payload of a `CONTINUE` frame MUST be empty.

7.1.4 IN_STREAM Frame

The `IN_STREAM` frame carries a value belonging to the input stream of a streaming-input RPC (methods for which **HasInputStream** is `true`). Clients MAY send zero or more `IN_STREAM` frames after receiving `CONTINUE`, as long as the method definition allows so.

The payload of an `IN_STREAM` frame consists of:

- A serialized stream element value, encoded using the type declared in the method signature.

Clients MUST NOT send `IN_STREAM` frames after sending `IN_CLOSE`, nor after sending `CANCEL`.

Sending `IN_STREAM` for a method that does not declare an input stream is a protocol error.

7.1.5 IN_CLOSE Frame

The `IN_CLOSE` frame signifies that the client has finished sending all elements of the input stream.

If **HasInputStream** is `true`, the client MUST send exactly one `IN_CLOSE` frame unless it cancels the RPC before normal completion. A `IN_CLOSE` frame MUST contain an empty payload.

Sending `IN_CLOSE` for a method that does not declare an input stream is a protocol error.

7.1.6 OUT_STREAM Frame

The `OUT_STREAM` frame carries a value belonging to the output stream of the RPC (methods for which **HasOutputStream** is `true`). Servers MAY send zero or more `OUT_STREAM` frames after sending `CONTINUE` and before sending `OUT_CLOSE`.

The payload of an `OUT_STREAM` frame consists of:

- A serialized stream element value, encoded with the declared output stream type.

Servers MUST NOT send `OUT_STREAM` frames after sending `OUT_CLOSE`, `CANCELLED`, or `ERROR`.

Sending `OUT_STREAM` for a method that does not declare an output stream is a protocol error.

7.1.7 OUT_CLOSE Frame

The `OUT_CLOSE` frame signifies that the server has sent all output stream elements.

A `OUT_CLOSE` frame MUST contain an empty payload and MUST be sent only once during the lifecycle of the RPC.

Sending `OUT_CLOSE` for a method that does not declare an output stream is a protocol error.

Servers MUST NOT send `OUT_CLOSE` frames after sending `CANCELLED`, or `ERROR`.

7.1.8 CANCEL Frame

The `CANCEL` frame is sent by the client to request early termination of an RPC.

A `CANCEL` frame has an empty payload.

Upon receiving `CANCEL` for an active RPC, the server:

- MUST notify the executor that the request has been cancelled.
- MUST send a single `CANCELLED` frame in response to the request.
- MUST NOT send any other frame after the `CANCELLED` frame.
- MUST consider any open stream as closed.
- MUST cause any attempt of sending a `RESPONSE`, `OUT_STREAM` to yield an error indicating that the RPC was cancelled.
- MUST cause any attempt of receiving a value through any available input stream to yield an error indicating that the RPC was cancelled.

RPCs MUST be aware of cancellation semantics, and must provide cancellation points to be able to correctly handle cancellation requests. Rollbacks, checkpoints, and other semantics related to cancellation are application-dependent and thus out of scope of this document.

A late `CANCEL` (one received after the server considers the RPC complete) MUST be ignored and MUST NOT trigger any additional frame.

7.1.9 CANCELLED Frame

The `CANCELLED` frame acknowledges receipt and processing of a client-issued `CANCEL` for a given `CorrelationID`. Its payload is empty; the `PayloadLength` field MUST be zero.

Upon sending `CANCELLED`, the server indicates that the RPC has been aborted at the client's request. A `CANCELLED` frame is a terminal, non-error outcome for that RPC: it terminates the RPC identified by its `CorrelationID` but does not imply that a fault occurred on the server. Other RPCs multiplexed on the same connection are unaffected.

An RPC terminated with a `CANCELLED` frame has all input and output streams, if any, implicitly closed. Servers MUST ensure that any internal buffer or queue is drained.

7.1.10 ERROR Frame

The `ERROR` frame is the terminal negative outcome of an RPC.

The payload of an `ERROR` frame MUST be an *error payload* structured as:

- A structured error body containing:
  - a machine-readable error code, and
  - a human-readable message string.
  - an optional arbitrary byte array containing extra diagnostic information provided by the application, if any applies. This field MUST not be interpreted by servers or libraries; the value is intended to only be relevant to the server and client implementations provided by the end-user.

An `ERROR` frame concludes the RPC for both endpoints. No further frames for the associated `CorrelationID` may be sent afterward.

7.1.11 RESPONSE Frame

The `RESPONSE` frame is the terminal positive output of an RPC.

The payload of an `RESPONSE` frame MUST be an *unary output tuple*. Even for frames containing no items, the first portion of the *unary output tuple* MUST be present, indicating a zero-length tuple, as specified in Section 6.6. For methods with unary outputs, the unary output tuple is encoded as the amount of items in the tuple through an VarUInt, followed by the serialized values of each of them.

7.2 Requests

An RPC request is initiated by the client sending an `INVOKE` frame. The on-wire layout of `INVOKE` (identifiers, and unary inputs, if any) is defined in Section 7.1

and its frame subsections. This subsection specifies only sequencing and lifecycle rules.

- For each `CorrelationID`, the client sends exactly one `INVOKE` as the first frame. The server responds with either `CONTINUE` (binding succeeded) or a terminal `ERROR` (binding failed).
- If **HasInputStream** is `true`, the input stream is considered open after `CONTINUE`. The client MAY send zero or more `IN_STREAM` frames. For a successfully completing RPC (no `CANCEL` or `ERROR`), the client MUST send exactly one `IN_CLOSE`, and the server MUST NOT send `RESPONSE` until it has received such `IN_CLOSE`. After `CANCEL`, the client MUST NOT send any further `IN_STREAM` or `IN_CLOSE` frames for that `CorrelationID`.
- If **HasOutputStream** is `true`, the output stream is considered open after `CONTINUE`. The server MAY send zero or more `OUT_STREAM` frames. For a successfully completing RPC (no `CANCEL` or `ERROR`), the server MUST send exactly one `OUT_CLOSE`, and the server MUST NOT send `RESPONSE` until it has sent such `OUT_CLOSE`. After `CANCELLED`, the server MUST NOT send any further `OUT_STREAM` or `OUT_CLOSE` frames for that `CorrelationID`.
- A client MUST NOT send another `INVOKE` with the same `CorrelationID`. Doing so is a protocol error. If the server receives a duplicate `INVOKE` for an active RPC, it MUST treat this as a connection-level protocol error and MUST close the transport without attempting to report it via `ERROR`.

7.3 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.

Frames, as defined in Section 7.1, are logical protocol units and do not necessarily align with transport-level segmentation. Implementations MUST tolerate arbitrary segmentation of frames by the underlying transport and MUST reconstruct complete frames from the incoming byte stream. Bytes belonging to different frames MUST NOT be interleaved on the wire (see Section 7.1).

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.

7.4 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.

7.4.1 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 7.8.2 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` as a protocol error, *except* for `CANCEL` frames. A `CorrelationID` is considered inactive if and only if the completion conditions in Section 7.8.2 have been satisfied for that identifier.

A `CANCEL` frame received for an unknown or inactive `CorrelationID` (as defined in Section 7.8.2) MUST be silently ignored and MUST NOT be treated as a protocol violation.

7.4.2 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`. A second `INVOKE` for an active `CorrelationID` is a connection-level protocol error (see Section 7.2 and Section 7.10.1).
- 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 7.8.2 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.

7.4.3 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.

7.5 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.

7.5.1 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.

7.6 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 7.4.

Frames considered as *invalid* per Section 5.7.2 are not included in Table 1.

Table 1: arf Method Forms and Permitted Streaming Frames

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
NYYN	N	Y	Y	N	Y	Y	N	N
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
YYYN	Y	Y	Y	N	Y	Y	N	N

Note: The permissions in this table apply only to RPCs that complete successfully. If an RPC is terminated by `CANCEL` or `ERROR`, `IN_CLOSE` and `OUT_CLOSE` are neither required nor permitted. Cancellation semantics are defined in Section 7.10.2.

Note: The table indicates which frame types may appear for a given method form. It does not express sequencing or success-completion requirements. When a stream is present, exactly one corresponding `IN_CLOSE` or `OUT_CLOSE` frame is required for successful completion (see Sections 7.2 and 7.8.4).

Legend

`HasIn`	HasInput
`HasOut`	HasOutput
`HasInS`	HasInputStream
`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`

The following rules apply only to RPCs that complete successfully. In the presence of cancellation or error, the stream-termination requirements in this section are overridden by the cancellation and error semantics defined in Section 7.10.

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

- The generic request invariants from Section 7.2 apply unchanged: for each `CorrelationID`, the client sends exactly one `INVOKE` as the first frame, and

the server responds with either `CONTINUE` (binding succeeded) or a terminal `ERROR` (binding failed).

- If HasInS is `Y` and the RPC initializes successfully, the client MAY send zero or more `IN_STREAM` frames.

  For any RPC that completes successfully (that is, with a `RESPONSE` and without `CANCEL`, `CANCELLED`, or `ERROR`), the client MUST send exactly one `IN_CLOSE` for that `CorrelationID`, and the server MUST NOT send `RESPONSE` until it has received that `IN_CLOSE`.

  If the RPC is terminated by `CANCEL`, `CANCELLED`, or `ERROR`, `IN_CLOSE` MUST NOT be sent. If HasInS is `N`, the client MUST NOT send `IN_STREAM` or `IN_CLOSE` at any time.

- If HasOutS is `Y` and the RPC is initialized successfully, the server MAY send zero or more `OUT_STREAM` frames.

  For any RPC that completes successfully, the server MUST send exactly one `OUT_CLOSE` frame for that `CorrelationID`, and it MUST send that `OUT_CLOSE` before sending `RESPONSE`.

  If the RPC is terminated by `CANCEL`, `CANCELLED`, or `ERROR`, the server MUST NOT send `OUT_CLOSE`. Cancellation or error implicitly closes the output stream. If HasOutS is `N`, the server MUST NOT send `OUT_STREAM` or `OUT_CLOSE` at any time.

Sending a `RESPONSE` frame for a method form with **HasInputStream** or **HasOutputStream** set to `true` *before* the required `IN_CLOSE` and/or `OUT_CLOSE` frames have been sent and received (in the successful, non-error, non-cancel case) is a protocol error. Because such an error cannot be reliably reported using an `ERROR` frame scoped to the same `CorrelationID`, implementations SHOULD treat it as a connection-level protocol violation and close the underlying transport.

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

The `INVOKE` frame layout (identifiers and optional unary inputs) is defined in Section 7.1. This subsection states only when the unary input tuple is present: if **HasInput** is `true`, the unary input tuple is included after the identifiers block; if **HasInput** is `false`, an empty unary input tuple is encoded. Clients MUST encode a zero-length input tuple even when `HasInput` is `false`.

7.8.2 RPC Completion

An RPC is considered to have completed *successfully* if and only if it terminates via `RESPONSE`.

A `RESPONSE` frame is a terminal condition only when it is validly transmitted, meaning:

- if **HasInputStream** is `true`, the client has already sent exactly one `IN_CLOSE` for that `CorrelationID`; and
- if **HasOutputStream** is `true`, the server has already sent exactly one `OUT_CLOSE` for that `CorrelationID`.

A `RESPONSE` received in violation of these sequencing rules MUST be treated as a protocol error and MUST NOT be considered successful completion.

Any RPC whose lifecycle includes a terminal `ERROR` frame is considered to have completed abnormally, while a terminal `CANCELLED` indicates that the RPC was cancelled by the client. The semantics of cancellation on the view-point of RPC method implementations is implementation-dependent and out of scope of this document. The only normative direction in cancellation is that the server MUST NOT send other frames for the cancelled `CorrelationID`, and MUST employ ways to notify the implementation that the RPC has been cancelled, as specified in Section 7.1.8 and 7.10.2.

For an RPC that is still active at the server, a client-initiated cancellation immediately aborts the execution of the RPC. The server MUST send a single `CANCELLED` frame, and cease sending any other frames for that RPC. More information is contained within Section 7.10.2.

For a given `CorrelationID`, an RPC is considered complete, from the point of view of a given endpoint, when that endpoint has observed one of the following terminal conditions for that endpoint:

- The endpoint has sent or received an `ERROR` frame for that `CorrelationID`; or
- The endpoint has observed either:
  - successful unary completion through a `RESPONSE` frame; or

  - cancellation through a `CANCELLED` frame.

Unary completion is directional:

- From the **client**'s perspective, unary completion occurs when it has *received* a `RESPONSE` frame for the `CorrelationID`.
- From the **server**'s perspective, unary completion occurs when it has *sent* a `RESPONSE` frame for the `CorrelationID`.

Stream closure is also directional:

- A stream flowing *from the client to the server* (the input stream when **HasInputStream** is `true`) is considered closed:
  - by the **client** when it has *sent* exactly one `IN_CLOSE` frame; and
  - by the **server** when it has *received* exactly one `IN_CLOSE` frame.
  - by both if any `ERROR` or `CANCELLED` frame is exchanged.
- A stream flowing *from the server to the client* (the output stream when **HasOutputStream** is `true`) is considered closed:
  - by the **server** when it has *sent* exactly one `OUT_CLOSE` frame; and
  - by the **client** when it has *received* exactly one `OUT_CLOSE` frame.
  - by both if any `ERROR` or `CANCELLED` frame is exchanged.

Thus, for a *successful* RPC (no `ERROR` or `CANCELLED` frame), the client considers the RPC complete when it has received `RESPONSE`. The client MUST consider the RPC as cancelled when a `CANCELLED` frame is received, and MUST consider it failed in case an `ERROR` frame is received.

Symmetrically, the server considers the RPC terminated when it has sent either `RESPONSE`, `ERROR`, or `CANCELLED`.

A `CANCEL` frame is not itself a terminal condition; it is a control signal that requests termination of the RPC and triggers the error semantics described in Section 7.10.2 and Section 7.10.1.

Once any of the above terminal conditions is satisfied for an endpoint, that endpoint MUST treat the corresponding `CorrelationID` as inactive and MUST NOT send any further frames with that `CorrelationID`.

With the exception of late `CANCEL` frames (see Section 7.10.2), any frame received for an inactive `CorrelationID` MUST be treated as a protocol error and MUST cause the connection to be closed.

Endpoints MUST NOT reuse a `CorrelationID` for a new RPC while they still consider the existing RPC for that identifier to be active. In particular, sending or receiving a `CANCEL` frame does not, by itself, render an RPC complete. A client that has sent `CANCEL` MUST NOT reuse the corresponding `CorrelationID` on that connection until it has either observed an `CANCELLED` frame, or the appropriate combination of `RESPONSE` and stream closures or closed the underlying transport connection.

7.8.3 Unary Output

The on-wire layout of `RESPONSE` is defined in Section 7.1. For method forms without an output stream (**HasOutputStream** = `false`), every RPC that completes successfully (no `ERROR`) MUST send exactly one `RESPONSE` for the corresponding `CorrelationID`, and it MUST be the last application-level frame for that RPC. All ordering and closure rules remain unchanged:

- If **HasOutput** is `true`, the `RESPONSE` payload contains the unary output tuple with encoded response values; otherwise, it contains an empty unary output tuple. In all legal method forms, successful completion is signalled by exactly one `RESPONSE` frame.
- If **HasOutputStream** is `true`, exactly one `OUT_CLOSE` MUST be sent before `RESPONSE`. No `OUT_STREAM` or `OUT_CLOSE` frames may follow `RESPONSE`.
- If **HasInputStream** is `true`, the server MUST NOT send `RESPONSE` until it has received exactly one `IN_CLOSE` for that `CorrelationID`.
- If the RPC terminates via `CANCELLED` or `ERROR`, input and output streams are implicitly closed; `IN_CLOSE` and `OUT_CLOSE` MUST NOT be sent afterward, and `RESPONSE` is not sent.

7.8.4 Output Stream

If **HasOutputStream** is `true`, the server MAY produce an output stream *after* sending the `CONTINUE` 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.
- If the RPC completes successfully (that is, with a `RESPONSE` and without `CANCEL` or `ERROR`), the server MUST send exactly one `OUT_CLOSE` frame for that `CorrelationID`, and it MUST send this `OUT_CLOSE` before the `RESPONSE` frame.

  If the RPC terminates due to `CANCEL` or `ERROR`, the server MUST NOT send `OUT_CLOSE`. Cancellation or error implicitly closes the output stream.
- The server MUST NOT send any `OUT_STREAM` or `OUT_CLOSE` frames after 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:

```
CONTINUE
OUT_STREAM*   // zero or more times
OUT_CLOSE
RESPONSE
```

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)
CONTINUE(CID)
RESPONSE(CID)
```

7.9.2 Form: NNNY (server-streaming only)

```
INVOKE(CID)
CONTINUE(CID)
OUT_STREAM(CID)* // zero or more items
OUT_CLOSE(CID)
RESPONSE(CID)
```

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

```
INVOKE(CID, unary-input)
CONTINUE(CID)
IN_STREAM(CID, T)* // zero or more items
```

```
IN_CLOSE(CID)
RESPONSE(CID)   // empty unary output tuple
```

Note: In the successful case (no `CANCEL` or `ERROR`), the client MUST eventually close the input stream by sending exactly one `IN_CLOSE` frame, and the server MUST NOT send `RESPONSE` until it has received that `IN_CLOSE`. The server is free to process input elements and decide its unary result as soon as it has sufficient information, but it is not permitted to complete the RPC on the wire (by sending `RESPONSE`) before the input stream has been explicitly closed.

7.9.4 Form: YYNN (Unary only)

```
INVOKE
CONTINUE
RESPONSE   // unary output tuple
```

7.9.5 Form: NNNY (output stream)

```
INVOKE
CONTINUE
OUT_STREAM* // zero or more items
OUT_CLOSE
RESPONSE   // empty unary output tuple
```

7.9.6 Form: YNYY (unary input, input stream, output stream, no unary output)

```
INVOKE (unary input tuple)
CONTINUE
IN_STREAM*   // zero or more items
OUT_STREAM*  // zero or more items
IN_CLOSE
OUT_CLOSE
RESPONSE   // empty unary output tuple
```

Note: As stated in other sections, `IN_STREAM` and `OUT_STREAM` may be freely interleaved; consuming and emitting items to both streams are left as a client and RPC method implementation detail that is out of scope for this specification.

7.10 Error and Cancellation Semantics

7.10.1 Error Frames

An `ERROR` frame represents terminal failure or non-successful termination of an RPC for the associated `CorrelationID`. For a given `CorrelationID`:

- At most one `ERROR` frame MAY be sent per endpoint for a given RPC.
- Once an endpoint has sent an `ERROR` frame, it MUST NOT send any further frames with that `CorrelationID`.
- Once an endpoint has received an `ERROR` frame, it MUST treat the RPC as terminated and MUST ignore any subsequent frames *for that RPC* with that `CorrelationID`.

Unless explicitly stated otherwise, any violation of the framing, sequencing, or typing rules defined in this specification is a protocol error. Protocol errors are connection-scoped conditions: the receiving endpoint MUST treat them as connection-level failures and MUST close the underlying transport connection. Only application-defined failures explicitly represented as `ERROR` payloads are RPC-scoped and MAY leave the connection itself intact.

The RPC error descriptor has the following logical structure:

```
struct RPCError {
    code    uint32;
    message string;
    details optional<bytes>;
}
```

The fields have the following semantics:

- `code` is a numeric error code. This specification reserves the following well-known values:
  - `0`: **UNKNOWN**. An unspecified server-side failure occurred.
  - `1`: **UNKNOWN_METHOD**. The RPC was rejected because the server could not find a method to bind the call.

  Additional code values MAY be defined by implementations and application profiles.

- `message` is a human-readable diagnostic string intended for logging and debugging. It MUST NOT be relied on for programmatic behavior.
- `details` is an optional opaque blob that MAY carry application-specific structured information (for example, a serialized domain-specific error structure). Its interpretation is entirely out-of-band and MAY vary between deployments.

Implementations SHOULD attempt to decode any non-empty `ERROR` payload as `RPCError`. If decoding fails, the receiver MUST still treat the `ERROR` frame as a terminal condition for the RPC and MAY surface a generic error to the application.

The existence, absence, or contents of an `ERROR` payload MUST NOT affect the framing, completion, or cancellation rules defined elsewhere in this specification.

The `ERROR` frame is scoped to a single RPC identified by its `CorrelationID`. Some classes of protocol violations are therefore not safely expressible via `ERROR`. In particular, if an endpoint receives a second `INVOKE` frame for a `CorrelationID` that it still considers active, it cannot safely emit an `ERROR` for that `CorrelationID` without risking misassociation with the original RPC. Such conditions MUST be treated as connection-level protocol errors: implementations MUST close the transport connection and MUST NOT attempt to report them using `ERROR`.

7.10.2 Cancellation

A client MAY request an ongoing RPC to be aborted using a `CANCEL` frame with the same `CorrelationID` as the active call. A `CANCEL` frame MUST carry an empty payload; its `PayloadLength` field MUST be zero.

Cancellation is scoped to a single RPC. A `CANCEL` frame affects only the RPC identified by its `CorrelationID`; it MUST NOT alter the state or progression of any other active RPCs multiplexed on the same connection.

Upon receiving a `CANCEL` frame for an *active* RPC (that is, one for which the completion conditions in Section 7.8.2 have not yet been satisfied), the server must notify the executor of the intention of the client to cancel the RPC. The executor then SHOULD handle the signal accordingly.

Once a `CANCEL` frame is received for an *active* RPC, the server MUST respond with a single `CANCELLED` frame, and cease sending any other frame for the same `CorrelationID`. Any RPC's attempts to send new frames MUST be reported as an error to the executor.

If the server receives a `CANCEL` frame for a `CorrelationID` that it already considers inactive (because the RPC has completed according to Section 7.8.2), it MUST treat the `CANCEL` as a late control signal, MUST NOT send any additional frames (including `ERROR` or `CANCELLED`) in response, and MUST NOT treat the late `CANCEL` as a protocol error solely by virtue of its lateness.

A client that sends `CANCEL`:

- MUST NOT treat the RPC as terminated for local application purposes immediately after sending `CANCEL`. Cancellation is only confirmed after a `CANCELLED` frame is received.
- MUST NOT reuse the `CorrelationID` for a new RPC on that connection until it has either observed a terminal condition as defined in Section 7.8.2 or closed the underlying transport connection; and
- MUST ignore any subsequently received application-level frames until the terminal `CANCELLED` acknowledgement.

If a `CANCEL` frame is received for an RPC that has already completed, the receiver MUST ignore the frame and MUST NOT respond with an `ERROR` or `CANCELLED`.

Sending `CANCELLED` implicitly closes the input stream and/or output stream, if any are currently open. No `IN_CLOSE` or `OUT_CLOSE` frames are required or permitted after cancellation.

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 SHOULD 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)

hex-value         = "Ox" 1*(DIGIT / HEXDIG)

; TYPES

type              = primitive-type / composite-type / plain-type

plain-type        = *(snake-ident ".") camel-ident / primitive-type

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      = camel-ident / integer-primitive / "string"

stream            = "stream" 1*WSP plain-type

primitive-type    =
      "bool"
    / "string"
    / "bytes"
    / integer-primitive
    / "float32" / "float64"
```

```
    / "timestamp"

integer-primitive =
    "int8"  / "int16"  / "int32"  / "int64"
    / "uint8" / "uint16" / "uint32" / "uint64"

; 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
                    [ "(" *WSP [annotation-params] *WSP ")" ]
                    *WSP *c-nl

annotation-params = annotation-param *(*WSP "," *WSP 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-disc         = hex-value / "0" / %x31-39 *DIGIT

enum-value        = annotations *CRLF *WSP
                    screaming-snake-ident *WSP
                    "=" *WSP enum-disc 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 plain-type
                     *(*WSP "," *WSP snake-ident 1*WSP plain-type)
                     [*WSP "," stream])

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

; NOTE: All plain-type occurrences in service-rpc-input and service-rpc-returns
; MUST resolve, after name resolution, to user-defined struct or enum types.
; Builtin primitive or composite types MUST NOT appear directly in RPC method
```

```
; signatures; see the Types and Methods sections of the core specification.
```

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 be treated as protocol errors: they MUST result in termination of the associated `CorrelationID` and MUST cause the underlying connection to be closed, as described in Section 7.10.1.

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

```c
#define ARF_FNV1A32_OFFSET 0x811C9DC5u
#define ARF_FNV1A32_PRIME  0x01000193u

/* Compute FNV-1a over an arbitrary byte sequence. */
uint32_t arf_fnv1a32(const uint8_t *data, size_t len) {
    uint32_t h = ARF_FNV1A32_OFFSET;

    for (size_t i = 0; i < len; i++) {
        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.

```c
/* Encode signed 32-bit integer into ZigZag form. */
static inline uint32_t arf_zigzag32_encode(int32_t n) {
    return (uint32_t)(((uint32_t)n << 1) ^ (uint32_t)(n >> 31));
}

/* Decode ZigZag-encoded 32-bit integer. */
static inline int32_t arf_zigzag32_decode(uint32_t z) {
    return (int32_t)((z >> 1) ^ (uint32_t)-(int32_t)(z & 1u));
}

/* Encode signed 64-bit integer into ZigZag form. */
static inline uint64_t arf_zigzag64_encode(int64_t n) {
    return (uint64_t)(((uint64_t)n << 1) ^ (uint64_t)(n >> 63));
}

/* Decode ZigZag-encoded 64-bit integer. */
static inline int64_t arf_zigzag64_decode(uint64_t z) {
    return (int64_t)((z >> 1) ^ (uint64_t)-(int64_t)(z & 1u));
}
```

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	0xFFFFFFFE	`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 FF 01`
INT64_MAX	0xFFFFFFFFFFFFFFFE	`FE FF 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 7.8.2 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 7.5.

E.7 Error Mapping and Diagnostics

arf defines a structured error payload format for `ERROR` frames via the `RPCError` descriptor (Section 7.10.1). This structure provides a machine-readable error code, a human-readable message, and optional opaque details.

Implementations SHOULD define internal error categories and map them onto `RPCError` as follows:

- transport-level failures (e.g., I/O errors, broken connections) SHOULD generally result in connection closure rather than an `ERROR` frame, as the framing layer is no longer reliable;
- framing- and decoding-level errors that occur *after* a valid frame boundary has been established MAY be mapped to an `ERROR` frame if the connection remains otherwise usable;
- application-level failures SHOULD be reported via structured `RPCError` values with implementation-defined codes and messages.

Implementations SHOULD use the `RPCError.details` field sparingly and only for optional, diagnostic data that is safe to expose. This field MAY carry structured, application-specific error information encoded out of band, but no interoperability is implied unless both peers explicitly agree on its format.

Error messages are intended for human diagnostics and logging only and MUST NOT be relied upon for programmatic behavior.

If an `ERROR` frame is received with an unparseable or malformed payload, the receiver MUST still treat the frame as terminal for the RPC and MAY surface a generic failure to the application.