Independent Submission
Internet-Draft
Intended status: ExperimentalCentre for Development of Advanced Computing (CDAC)
Expires: 13 June 2026 10 December 2025

A. Mallick
I. Chebolu

The Micro Agent Communication Protocol (袖ACP)
draft-mallick-muacp-00

Abstract

This document specifies the Micro Agent Communication Protocol (袖ACP), a resource-efficient messaging protocol for autonomous agents operating on constrained devices (Class 1 IoT devices per [RFC7228]). Existing agent communication protocols assume unbounded computational and energy resources; 袖ACP provides formal guarantees on memory, energy, and bandwidth consumption while maintaining expressiveness sufficient for finite-state coordination patterns. The protocol defines four core message types, a fixed 64-bit header, TLV-based extensibility, and mandatory OSCORE security binding for operation in adversarial environments.

Status of This Memo

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

This Internet-Draft will expire on 13 June 2026.

Copyright Notice

Table of Contents

1. Introduction

The Micro Agent Communication Protocol (ツオACP) is a compact, resource-efficient communication protocol designed for distributed autonomous agents operating on constrained devices. It aims to bridge the gap between resource-light IoT protocols and semantically rich agent communication languages, by offering minimal overhead yet expressive interaction semantics.

Modern IoT, edge, and embedded environments often involve devices with limited RAM, CPU, energy, and unreliable or low-bandwidth networks. At the same time, many distributed applications 寰from sensor networks and robotics swarms to multi-agent systems and edge-native microservices — require coordination, state sharing, event subscriptions, request/response semantics, and lightweight negotiation. Existing protocols are often unsuited:

* Traditional agent-communication languages (e.g., FIPA-ACL) impose heavy parsing and runtime overhead unacceptable on microcontroller-class platforms.

* Standard IoT protocols (e.g., plain UDP, lightweight publish/subscribe) provide minimal semantics, making it difficult to implement structured coordination or stateful dialogues.

ACP addresses this by defining a wire-efficient, fixed-header, TLV-extensible protocol that offers exactly four core verbs — PING, TELL, ASK, and OBSERVE — which together are sufficient to express common interaction patterns such as request/response, publish/subscribe, and liveness checking. The protocol is designed so that implementations can remain lean, deterministic in resource consumption, and suitable for microcontroller-class devices, while still supporting structured multi-agent interactions.

Because security, confidentiality, and integrity are essential for many deployments (especially those involving sensitive data, distributed control, or untrusted networks), this specification mandates the use of the object-security mechanism defined by the IETF as the mandatory-to-implement transport binding: namely, the combination of CoAP (as the transport substrate) with OSCORE (for application-layer message protection) over constrained or lossy links. This ensures that even devices with limited resources can securely exchange ACP messages while preserving end-to-end confidentiality, integrity, and replay protection [RFC8613].

Working Group Engagement: This document is submitted as an Independent Submission to the IETF. The authors welcome feedback from relevant working groups, particularly the Constrained RESTful Environments (CoRE) working group and the Light-Weight Implementation Guidance (LWIG) working group, and are open to transitioning this work to a working group if there is community interest and consensus.

1.1. Goals

* Provide a minimal, low-overhead communication protocol for constrained agents that supports structured semantics without heavy runtime cost.

* Ensure deterministic and bounded resource usage (memory, CPU, bandwidth), enabling predictable behavior in resource-constrained environments.

* Support essential multi-agent communication patterns — request/response, publish/subscribe, life-check — using a small set of orthogonal primitives.

* Define a secure, interoperable transport binding so that agents across different platforms can communicate safely and reliably.

* Enable extensibility via a TLV option mechanism, allowing future enhancements (e.g., content types, metadata, authentication tokens) without breaking base compatibility.

1.2. Scope

This specification defines the wire format, core semantics, normative behavior, mandatory transport binding, security constraints, and IANA registries required for interoperable implementations. It does not specify or mandate application-level semantics (e.g., content encoding, agent ontology, high-level negotiation logic), which are left to deployment-specific or higher-layer protocols. Implementers are free to choose content encoding (e.g., CBOR, JSON), TLV usage, and higher-level behavior, provided they adhere to the normative parts of this document.

1.3. Intended Audience

This document is primarily intended for:

* Developers of embedded, IoT, or edge-device firmware seeking a lightweight yet expressive agent communication protocol;

* Protocol engineers designing distributed multi-agent systems requiring structured interactions, event subscriptions, or resource-aware communication;

* Standardization bodies and implementers evaluating ACP for integration into larger systems;

* Researchers studying resource-bounded multi-agent coordination, secure constrained communication, or constrained-device protocol design.

1.4. Relation to Existing Work

ACP’s transport binding leverages established IETF standards: CoAP as the constrained-device transport substrate [RFC7252], and OSCORE for end-to-end object security across constrained networks and proxies [RFC8613]. OSCORE is itself designed for constrained RESTful environments and provides confidentiality, integrity, and replay protection for CoAP messages using COSE [RFC8949].

Compared to heavier agent-communication languages (e.g., FIPA-ACL), ACP trades semantic depth for minimalism and resource efficiency. Compared to plain CoAP or MQTT-style protocols, ACP brings structured agent-oriented primitives while preserving low overhead and deterministic resource usage. The TLV extensibility mechanism ensures future content- or metadata-level enhancements without invalidating base interoperability.

1.5. Document Structure

The remainder of this document is organized as follows:

- Section 2 defines conventions and terminology.

- Section 3 describes the ACP message model and wire encoding rules.

- Section 4 defines the protocol semantics of the four core verbs (PING, TELL, ASK, OBSERVE).

- Section 5 defines the mandatory transport binding using CoAP/ OSCORE, including mapping rules, security profile, and operational constraints.

- Section 6 defines error handling rules, version negotiation, and extensibility mechanisms.

- Section 7 defines the IANA registries (TLV Types, QoS codes, content-format) and related registration policies.

- Section 8 provides normative interoperability requirements and deployment guidelines.

- Section 9 describes security considerations and threat mitigation strategies.

- Section 10 provides example interactions and wire-level encodings.

- Appendices contain deployment notes, conformance test descriptions, change log, and reference implementation pointers.

2. 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 BCP 14 [RFC2119] [RFC8174] when they appear in ALL CAPS. These words may also appear in lowercase or mixed case as plain English words, absent their normative meanings.

2.1. Terminology

The following terms are used throughout this document:

Agent An autonomous software entity that participates in ACP communication. Agents send and receive ACP messages to coordinate with other agents.

Verb One of four core communication primitives in ACP: PING, TELL, ASK, or OBSERVE. Verbs are encoded in 2 bits in the message header.

TLV Type-Length-Value encoding format used for optional metadata in ACP messages. Each TLV consists of an 8-bit Type, 8-bit Length, and variable-length Value.

Correlation ID A 16-bit identifier that groups related messages into a conversation. Messages sharing the same Correlation ID belong to the same conversation.

Sequence ID A 16-bit monotonically increasing identifier used for duplicate detection and replay protection within a conversation.

Conversation A sequence of related ACP messages identified by a unique Correlation ID. Conversations typically represent request/response exchanges or subscription relationships.

OSCORE Object Security for Constrained RESTful Environments, as defined in [RFC8613]. OSCORE provides end-to-end security for CoAP messages.

CoAP The Constrained Application Protocol, as defined in [RFC7252]. CoAP serves as the transport substrate for ACP.

Constrained Device A device with limited resources (memory, CPU, energy, bandwidth) as defined in [RFC7228]. Class 1 devices have approximately 10 KB RAM and 100 KB flash.

2.2. Notation

This document uses the following notation conventions:

* Hexadecimal values are prefixed with "0x" (e.g., 0x01, 0xFF).
* Binary values are prefixed with "0b" (e.g., 0b00, 0b11).
* Byte order is network byte order (big-endian) unless otherwise specified.
* Bit positions are numbered from 0 (most significant bit) to n-1 (least significant bit).
* Message formats are shown using ASCII art diagrams.
* Code examples use a monospace font.

2.3. Abbreviations

The following abbreviations are used in this document:

ACL Agent Communication Language

BDI Belief-Desire-Intention (agent architecture model)

CBOR Concise Binary Object Representation [RFC8949]

CID Correlation ID

CoAP Constrained Application Protocol

COSE CBOR Object Signing and Encryption

DTLS Datagram Transport Layer Security

EDHOC Ephemeral Diffie-Hellman Over COSE [RFC9528]

FSM Finite State Machine

IANA Internet Assigned Numbers Authority

IoT Internet of Things

MTI Mandatory to Implement

OSCORE Object Security for Constrained RESTful Environments

QoS Quality of Service

SID Sequence ID

TLV Type-Length-Value

URI Uniform Resource Identifier

3. Message Model and Encoding Rules

This section defines the normative wire-level encoding of ACP messages, including the fixed header, TLV format, payload processing rules, byte ordering, and OSCORE protection boundaries. All compliant implementations MUST follow these encoding rules exactly unless otherwise specified.

3.1. Message Structure

A ACP message consists of three components encoded in the following order:

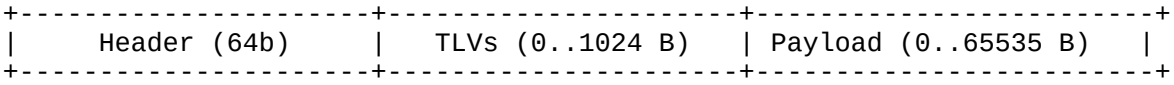

Figure 1: Figure 1: ACP Message Layout

The header format is fixed-length and MUST always appear. TLVs and payloads are optional. Messages MUST NOT exceed transport-imposed size limits; for CoAP/OSCORE, these limits are determined by underlying MTU constraints and CoAP Blockwise Transfer (RFC 7959) if used.

All fields are encoded in network byte order (big-endian).

3.2. Header Format

The ACP header consists of 64 bits arranged as follows:

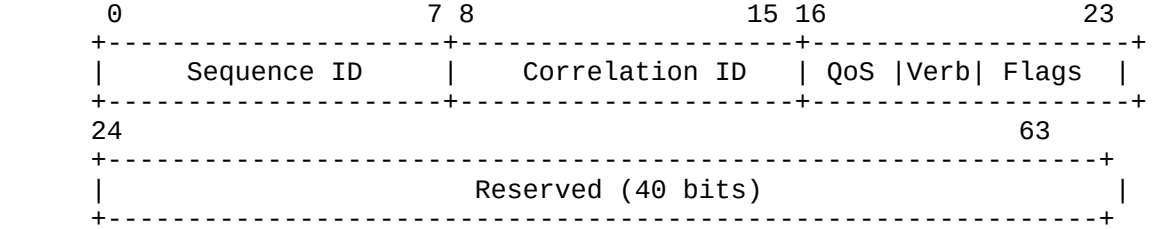

Figure 2: Figure 2: ACP Header Bit Layout

Sequence ID (16 bits): Monotonically increasing identifier used for duplicate detection and replay-window tracking. MUST wrap modulo 2^16. MUST be unpredictable if security-sensitive traffic requires preventing traffic analysis.

Correlation ID (16 bits): Identifies all messages belonging to the same conversation. MUST be unique among active conversations. SHOULD be randomly generated in security-sensitive deployments.

QoS (2 bits): Encodes transmission semantics (fire-and-forget, at-least-once, at-most-once). Values are defined in the IANA Considerations section.

Verb (2 bits): Identifies one of the four ACP operations: PING(0), TELL(1), ASK(2), OBSERVE(3).

Flags (4 bits): Control bits reserved for protocol-level features such as fragmentation, retransmission hints, or message cancellation. Future specifications MAY define additional meanings.

Reserved (40 bits): MUST be set to zero on transmission. MUST be ignored by receivers. Reserved bits MAY be repurposed by future ACP versions but MUST NOT change meaning in this version.

3.3. TLV Encoding

TLVs (TypeLengthValue structures) convey optional metadata and extensibility information. They appear immediately after the header and MUST appear in Type-increasing order to allow binary search and deterministic parsing.

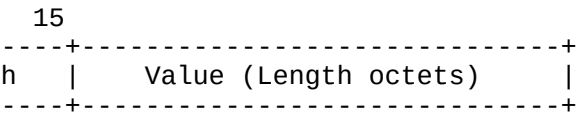

Figure 3: Figure 3: TLV Encoding

Type (8 bits): TLV identifier. The meaning of each Type is defined in the IANA registry. Types 031 are reserved and governed by Standards Action. Types 32127 require IETF Review. Types 128255 are vendor-specific.

Length (8 bits): Specifies the number of octets in the Value field. Length MUST NOT exceed 255. TLVs MUST NOT cause the total TLV region to exceed 1024 bytes.

Value: Encoded according to Type. For Types other than 0x00 (Raw Octets), the Value is subject to OSCORE protection (Section 5).

Critical TLVs: A future TLV Type range MAY designate critical TLVs. Receiving an unknown critical TLV MUST cause message rejection.

3.3.1. TLV Processing Rules

Receivers MUST apply the following rules when processing TLVs:

* TLVs MUST be parsed strictly in order.
* If Length exceeds remaining buffer size, the message MUST be discarded.
* Unknown TLV Types MUST be ignored unless they are designated critical.
* TLV order MUST be strictly increasing by Type; violating this is a format error.
* TLV Type 0x00 (Raw Octets) MUST NOT appear in encrypted messages; its use is restricted to unencrypted PING messages.

3.4. Payload Encoding

The ACP payload is an optional octet string of 065535 bytes. Its semantics depend on the Verb and the application layer. Payloads are typically used for:

* Application data (e.g., sensor readings, state updates);
* Action parameters and operation descriptors;
* Event notifications for OBSERVE subscriptions;
* Encoded content (CBOR, JSON, application-specific formats).

Payloads MUST be OSCORE-protected unless the message Verb is PING. Payload sizes MUST be validated before allocation to avoid resource exhaustion.

If the payload is encoded using CBOR (Type=0x03), receivers MUST treat it as a single CBOR data item. If the payload is JSON (Type=0x02), it MUST be UTF-8 encoded.

3.5. Byte Ordering

All multi-octet integer fields in ACP (Sequence ID, Correlation ID, header composites) MUST be encoded in network byte order (big-endian). TLV and payload content MAY use other encoding rules (e.g., CBOR or UTF-8) as determined by their Types.

3.6. Fragmentation (Optional Feature)

ACP itself does not mandate fragmentation. When implemented, fragmentation MUST be controlled by Flags in the header and MUST operate in a deterministic, resource-bounded manner.

This specification defines the following fragmentation requirements:

* Fragments MUST preserve the same Sequence ID, Correlation ID, and Verb.

* Fragments MUST carry a Fragment-ID TLV if fragmentation is used.

* Receivers MUST reassemble fragments in Sequence ID order.

* Reassembly MUST abort if missing fragments exceed timeout thresholds.

Deployments using CoAP Blockwise Transfer (RFC 7959) SHOULD avoid ACP-level fragmentation to minimize complexity.

3.7. OSCORE Protection Boundaries

When ACP is transported over CoAP with OSCORE, the OSCORE-protected CoAP payload MUST contain the complete ACP message (Header | TLVs | Payload). The OSCORE security context determines integrity, confidentiality, and replay parameters.

The following MUST be protected by OSCORE:

* All TLVs except those in unencrypted PING messages;

* The entire payload;

* The header fields other than those needed for outer CoAP routing.

Implementations MUST NOT leak semantics (e.g., Verb, QoS) through the CoAP outer header beyond what OSCORE permits.

3.8. Canonical Encoding Rules

To ensure interoperability and deterministic parsing, ACP defines the following canonical encoding rules:

* Fields MUST NOT be padded.

* TLVs MUST be sorted by ascending Type.

* No two TLVs MAY share the same Type unless explicitly defined.

* Payload MUST begin immediately after the last TLV.

* Implementations MUST normalize line endings, whitespace, or internal representations before hashing or signing application content.

These rules ensure that ACP messages can be compared as byte strings and efficiently parsed on constrained devices.

4. Protocol Semantics

This section defines the normative semantics of the four ACP verbs: PING, TELL, ASK, and OBSERVE. Each verb represents a fundamental communication primitive intended to support higher-level agent behaviors, including liveness detection, request/response interactions, state dissemination, and event-driven notification.

Agents MUST implement all four verbs. Agents MUST apply OSCORE protection to all messages except PING, unless an application explicitly operates in an unauthenticated environment.

For each verb, this section defines sender behavior, receiver behavior, state-machine interactions, mandatory error cases, and expected processing-time bounds.

4.1. PING

PING provides a low-cost, minimal-overhead mechanism for reachability and liveness detection. PING messages MUST NOT carry OSCORE-protected content. PING messages MAY include Raw-Octet TLVs (Type=0x00) but MUST NOT include any TLVs requiring confidentiality or integrity.

4.1.1. Sender Behavior

* The sender MAY emit a PING at any time to test peer liveness.

* The Sequence ID MUST increment for each PING from the same sender.

* The Correlation ID SHOULD be unique among active PING probes to avoid ambiguity.

* No payload is RECOMMENDED; the PING SHOULD remain as small as possible.

* PING SHOULD be rate-limited to avoid unnecessary resource pressure.

4.1.2. Receiver Behavior

* The receiver SHOULD reply with a TELL unless configured otherwise.

* The receiver MUST NOT require OSCORE protection for PING.

* The receiver MAY update local reachability or suspicion heuristics.

* The receiver MUST ignore any TLVs not allowed for PING.

4.1.3. Error Conditions

* If a PING contains encrypted TLVs or a payload, the receiver MUST treat the message as malformed.

* If resource limits are exceeded, the receiver MAY silently drop the PING.

4.2. TELL

TELL conveys information, updates, or asynchronous notifications. TELL is the primary mechanism for distributing state and for responding to ASK messages.

TELL messages MUST be OSCORE-protected unless the deployment explicitly allows an unauthenticated mode.

4.2.1. Sender Behavior

* The sender MUST include a payload or a meaningful TLV set.

* The sender SHOULD ensure that TELL messages are idempotent for the given Correlation ID.

* The sender MUST increment the Sequence ID for each TELL.

* When responding to an ASK, the sender MUST use the same Correlation ID as the request.

* If notifying subscribers (OBSERVE), the sender MUST include event-related TLVs.

4.2.2. Receiver Behavior

* The receiver MUST validate OSCORE before inspecting payload or TLVs.

* The receiver MUST associate the message with the correct conversation using Correlation ID.

* The receiver MUST incorporate content into its knowledge base according to application policy.

* The receiver SHOULD acknowledge or respond only if required by application semantics.

* In OBSERVE-driven notifications, the receiver MUST update subscription state accordingly.

4.2.3. Error Conditions

* If a TELL arrives without OSCORE protection, the receiver MUST reject it unless configured for non-secure operation.

* If a TELL carries malformed TLVs, the receiver MUST discard the message.

* If correlation state does not exist and the TELL is unsolicited, the receiver MAY treat it as a standalone notification.

4.3. ASK

ASK initiates a request for information or action. ASK is analogous to a query, command, or method invocation, and typically elicits a TELL response.

ASK messages MUST be OSCORE-protected.

4.3.1. Sender Behavior

* The sender MUST allocate a new conversation entry indexed by the Correlation ID.

* The sender MUST increment the Sequence ID.

* The ASK payload SHOULD include sufficient information for the receiver to satisfy the request.

* The sender MUST start a request timer; expiration triggers retransmission (for QoS 1) or failure escalation.

* The sender MUST enforce conversation limits to avoid resource exhaustion.

4.3.2. Receiver Behavior

* The receiver MUST validate OSCORE before processing.

* The receiver MUST associate the ASK with the given Correlation ID, creating state if needed.

* The receiver MUST generate a TELL response with either the requested result or an error TLV.

* If the request cannot be satisfied in bounded time, the receiver MAY send an immediate error TELL.

* If performing an action, the receiver SHOULD ensure bounded execution time or respond asynchronously.

4.3.3. Error Conditions

* If the ASK contains malformed or conflicting TLVs, the receiver MUST reject it using a TELL(error).

* If security validation fails, the receiver MUST silently discard the message.

* If correlation-table limits are exceeded, the receiver MAY respond with a resource exhaustion error.

4.4. OBSERVE

OBSERVE establishes a subscription for future notifications. It is analogous to a publish/subscribe registration but scoped to a single peer.

OBSERVE messages MUST be OSCORE-protected.

4.4.1. Sender Behavior

* The sender MUST allocate or update a subscription state entry indexed by Correlation ID.

* The sender MUST validate that subscription limits have not been exceeded.

* The sender MUST increment the Sequence ID.

* The sender MAY include TLVs expressing subscription parameters (e.g., topic, conditions).

* The sender MUST send periodic notifications (TELL) while the subscription remains active.

4.4.2. Receiver Behavior

* The receiver MUST validate OSCORE before processing.

* The receiver MUST establish or refresh subscription state.

* The receiver SHOULD acknowledge with a TELL containing subscription parameters.

* The receiver MUST enforce subscription expiration, backpressure rules, and resource ceilings.

* When conditions are met, the receiver MUST send event notifications as TELL messages.

4.4.3. Subscription Cancellation

Subscription cancellation is performed when a TELL or OBSERVE message carries a Cancel-Subscription TLV (Type=0xFF). Upon cancellation:

* The receiver MUST delete subscription state.

* The receiver MUST stop sending notifications.

* The receiver MAY send a TELL confirming deletion.

4.4.4. Error Conditions

* If subscription limits are exceeded, the receiver MUST reject the OBSERVE with a TELL(error).

* If the OBSERVE contains a topic or condition TLV not understood by the receiver, the receiver MAY reject it.

* If OSCORE validation fails, the message MUST be dropped.

4.5. Summary of Normative Requirements

The following summarizes the semantic requirements of each verb:

* *PING:* Liveness probe; MUST NOT require OSCORE; MUST NOT include protected TLVs.

* *TELL:* Update/response/notification; MUST use OSCORE except in explicitly insecure deployments.

* *ASK:* Request; MUST use OSCORE; MUST generate a TELL response.

* *OBSERVE:* Subscription; MUST use OSCORE; MUST create or update subscription state.

Agents MUST NOT overload verbs with incompatible semantics. All application-defined behaviors MUST build upon these primitives in a manner that preserves ACP' s resource and security guarantees.

5. Mandatory Transport Binding: OSCORE/CoAP

This section defines the mandatory-to-implement (MTI) transport binding for ACP: the combination of the Constrained Application Protocol (CoAP) as the transport substrate and OSCORE as the end-to-end object security mechanism. All compliant ACP implementations MUST support this binding.

Deployments MAY support additional bindings (e.g., DTLS/UDP or QUIC) but such bindings are outside the scope of this specification and MUST NOT weaken or replace the OSCORE/CoAP MTI profile.

5.1. Mapping ACP Messages to CoAP

Each ACP message (Header | TLVs | Payload) is encoded as a byte string and placed entirely within the CoAP message payload. Only OSCORE-protected CoAP messages may carry ACP messages (except PING, which MAY be unprotected under specific deployment configurations).

ACP messages MUST use the following CoAP message structure:

* *Method:* POST

* *URI-Path:* "muacp" (fixed path for interoperability)

* *Content-Format:* application/muacp+binary

* *Payload:* Full ACP message

This yields the canonical envelope:

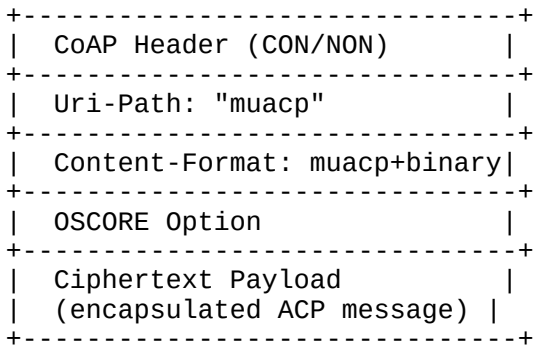

Figure 4: Figure 4: CoAP Envelope Carrying a ACP Message

Each ACP message corresponds to exactly one CoAP POST. For request/response interactions (ASK → TELL), CoAP confirmable/non-confirmable messages MAY be used depending on QoS requirements.

5.2. OSCORE Protection Requirements

All ACP messages except PING MUST be protected using OSCORE. OSCORE provides confidentiality, integrity, and replay protection independent of the transport layer.

OSCORE MUST protect the following elements:

* The entire ACP header (except when outer CoAP metadata is required for routing).

* All TLVs except raw TLVs permitted for PING.

* The entire ACP payload.

OSCORE replay protection MUST be enabled. Implementations MUST configure replay windows to match expected message rate and resource constraints.

OSCORE MUST use a unique security context per agent-pair. Context reuse between unrelated peers is prohibited.

5.3. Establishing OSCORE Security Contexts

Security contexts for OSCORE MAY be derived by any of the following methods:

* *EDHOC (RECOMMENDED):* A lightweight authenticated key exchange protocol suitable for constrained devices.

* *Pre-Shared Keys (PSK):* For deployments with pre-configured trust anchors.

* *Out-of-band provisioning:* Where security associations are established during manufacturing or commissioning.

When EDHOC is used, the resulting OSCORE context MUST be bound to the EDHOC handshake transcript to prevent identity misbinding attacks.

5.4. CoAP Message Types and Reliability

ACP builds upon CoAP reliability semantics to achieve its QoS model. Implementations MUST map ACP QoS codes to CoAP message types as follows:

ACP QoS	Meaning	CoAP Message Type
0	fire-and-forget	NON (Non-confirmable)
1	at-least-once delivery	CON (Confirmable)
2	at-most-once delivery	NON (No retransmission)

Table 1

CoAP-level acknowledgments MUST NOT be interpreted as ACP-level responses. Application responses are always encoded as TELL messages.

5.5. Mapping ASKTELL to CoAP Request/Response

ASK messages MUST be sent as CoAP POST requests. Corresponding TELL responses MUST be sent as CoAP responses. OSCORE MUST protect both directions.

The Correlation ID uniquely links the ASK with the TELL response. CoAP Message IDs MUST NOT be used for application correlation.

Receivers MUST respond with a TELL message even when requests fail, using an Error TLV to describe failure conditions.

Figure 5: Figure 5: ASK/TELL Over OSCORE-CoAP

5.6. Mapping OBSERVE Subscriptions

OBSERVE establishes a long-lived subscription. Subscriptions are maintained by application logic and do NOT rely on CoAP' s Observe extension (RFC 7641). ACP defines its own subscription model, independent of CoAP's Observe option.

OBSERVE MUST be mapped as:

* A CoAP POST containing an ACP OBSERVE message.

* Notification messages delivered as CoAP POSTs containing TELL messages.

Implementations MUST NOT use CoAP Observe for ACP subscriptions, to avoid semantic conflicts.

5.7. Congestion Control Requirements

All ACP-over-CoAP deployments MUST implement congestion control to prevent network collapse and unfair bandwidth usage.

Agents MUST adhere to the following rules:

* Apply exponential backoff on CoAP CON retransmissions.

* Rate-limit PING to avoid liveness floods.

* Throttle OBSERVE notifications when bandwidth pressure is detected.

* Maintain deterministic CPU and buffer usage for message handling.

* Avoid generating more than one message per RTT per conversation, except under QoS 1 retransmission.

When Blockwise Transfer (RFC 7959) is used, agents MUST ensure block sizes do not exceed memory limits.

5.8. Transport-Layer Error Handling

Transport errors such as CoAP timeouts, OSCORE decryption failures, or missing acknowledgments MUST be translated into ACP-level behavior rather than silently ignored.

Specifically:

* If OSCORE decryption fails, the ACP message MUST be dropped.
* If a CoAP CON message is not acknowledged, the sender MUST apply ACP QoS semantics to determine retransmission.
* Repeated timeouts MUST cause the ACP conversation to enter a failure state.
* Malformed CoAP envelopes MUST cause message discard.

5.9. Summary of MTI Requirements

All compliant ACP implementations MUST:

* Support CoAP POST requests to the fixed path "muacp".
* Support Content-Format: application/muacp+binary.
* Protect all messages except PING with OSCORE.
* Enforce OSCORE replay protection.
* Derive OSCORE contexts using EDHOC or equivalent secure provisioning.
* Map QoS codes to CoAP message types according to the CoAP Message Types and Reliability section.
* Generate TELL responses for all ASK messages.
* Deliver notifications for active OBSERVE subscriptions as TELL messages.

This binding ensures interoperability across all ACP implementations and establishes a minimum security baseline for deployments.

6. Error Handling, Version Negotiation, and Extensibility

This section defines the normative error-handling rules for ACP, the version-negotiation mechanism, downgrade protection requirements, and the extensibility framework provided by the TLV architecture. Proper handling of malformed messages, incompatible versions, and future extensions is essential for interoperability and robustness.

6.1. Error Code TLVs

All protocol-level errors MUST be communicated using a TELL message that includes an Error-Code TLV. Error codes are encoded as unsigned integers and MUST follow the registry defined in the IANA Considerations section.

Type: 0x22 (Error-Code, see IANA registry)
Length: 1 or 2 octets
Value: Integer error code

Figure 6: Error-Code TLV

The sender MUST set the Correlation ID of the error response to match the ID of the failing message. Receivers MUST interpret the error code as part of the ACP conversation state.

6.2. Standardized Error Conditions

The following error codes are defined for ACP:

Code	Name	Description
0x01	ERR_MALFORMED	Malformed header, TLV, or payload.
0x02	ERR_UNSUPPORTED_VERB	Verb not recognized or not supported by receiver.
0x03	ERR_UNSUPPORTED_TLV	Critical TLV not understood.
0x04	ERR_FORBIDDEN	Operation not permitted due to policy or authorization.
0x05	ERR_RESOURCE_EXHAUSTED	Memory, CPU, or subscription/ conversation limits exceeded.
0x06	ERR_VERSION_MISMATCH	Message uses unsupported protocol version.
0x07	ERR_TIMEOUT	Sender or receiver timed out while waiting for a response.
0x08	ERR_INTERNAL	Internal failure not covered by other error categories.

Table 2

Implementations MAY define additional vendor-specific error codes in the vendor range but MUST NOT redefine standardized codes.

6.3. Handling Malformed Messages

Receivers MUST apply strict validation to protect against malformed messages and resource attacks. Specifically:

* If TLV Length exceeds remaining bytes, the message MUST be discarded.

* If TLVs appear out of Type order, the message MUST be discarded.

* If a required TLV (future versions) is absent, the message MUST be rejected.

* If header fields contain invalid combinations (e.g., reserved bits set), the message MUST be rejected.

* If OSCORE decryption fails, the message MUST be discarded without error signaling.

Where feasible, a receiver SHOULD send a TELL(error) message to report failure, unless doing so would amplify a denial-of-service attack.

6.4. Conversation-Lifetime Error Handling

Conversations MAY fail due to timeouts, resource limits, or message corruption. When such failures occur:

* The agent MUST free associated resources (conversation-table entries).

* The agent SHOULD send an ERR_TIMEOUT or ERR_RESOURCE_EXHAUSTED TELL message.

* For resource exhaustion, an agent MUST NOT attempt recovery that risks violating its resource budget.

Conversations MUST be terminated when Correlation IDs collide.

6.5. Version Negotiation

ACP includes a Version field in the TLV space to allow forward compatibility. A Version-TLV (Type=0x01) MAY be included in any message to indicate the sender's supported protocol versions.

```
Type:   0x01 (Version)
Length: N (number of supported versions)
Value:  Array of version numbers (e.g., [0x00])
```

Figure 7: Figure 7: Version TLV

Receivers MUST ignore Version-TLVs indicating versions higher than supported. Receivers MUST accept messages labeled as version 0x00 (this specification) unless malformed.

If a message indicates only unsupported versions, the receiver MUST return ERR_VERSION_MISMATCH.

6.6. Downgrade and Version-Rollback Protection

Implementations MUST ensure that attackers cannot force a peer to use a lower protocol version when a higher mutually supported version is available.

Specifically:

* When a Version TLV lists multiple supported versions, the highest mutually supported version MUST be chosen.

* Version negotiation MUST occur inside OSCORE-protected messages except for PING.

* Agents MUST NOT downgrade versions unless a failure condition explicitly requires fallback.

6.7. Extensibility Framework

ACP is designed to evolve through extensible mechanisms based on TLVs. The following constraints ensure future versions remain interoperable:

* *Forward compatibility:* Receivers MUST ignore unknown non-critical TLVs.

* *Backward compatibility:* Implementations MUST NOT reuse TLV Types for different semantics.

* *Critical TLVs:* Future versions MAY introduce critical TLVs; receiving an unsupported critical TLV MUST trigger ERR_UNSUPPORTED_TLV.

* *TLV ordering:* All TLVs MUST be sorted by increasing Type value.

* *Vendor extensions:* Types 128255 are reserved for vendor-specific semantics and MUST NOT require global registration.

Complex or multi-field extensions SHOULD define new structured TLVs rather than overloading primitive types.

6.8. Summary of Normative Requirements

This section can be summarized as follows:

* Malformed messages MUST be rejected and SHOULD trigger a TELL(error) unless unsafe.

* Errors MUST use standardized codes where applicable.

* Version negotiation MUST prefer the highest mutually supported version.

* Unknown non-critical TLVs MUST be ignored; unknown critical TLVs MUST trigger errors.

* OSCORE failures MUST cause silent discard.

* Resource exhaustion MUST lead to conservative cleanup behavior.

These requirements ensure that ACP remains robust, extensible, and secure across diverse deployments.

7. IANA Considerations

This section requests the creation of new registries and assignments required for ACP to function as an interoperable Internet protocol. All registries use the policies defined in [RFC8126]. Unless otherwise stated, values are allocated using the "IETF Review" policy.

7.1. ACP TLV Types Registry

IANA is requested to create a new registry entitled "ACP TLV Types". The registry consists of 8-bit values (0255). Each entry MUST contain:

* Value (0255)

* Name

* Description

* Value format (e.g., integer, UTF-8, CBOR)

* Reference

The range is divided as follows:

* *031:* Standards Action

* *32127:* IETF Review

* *128255:* Vendor-specific

IANA is requested to populate the registry with the initial values below:

Value	Name	Description	Format	Reference
0x00	RAW_OCTETS	Unstructured data; MUST NOT appear in encrypted messages except PING.	Opaque	This document
0x01	VERSION	Advertised supported protocol versions.	Array of integers	This document
0x02	CONTENT_TYPE	Specifies payload encoding.	Integer	This document
0x03	CBOR_PAYLOAD	Payload encoded as CBOR.	CBOR data item	This document
0x20	TOPIC	Subscription topic for OBSERVE.	UTF-8 string	This document
0x21	CONDITION	Trigger condition for OBSERVE.	UTF-8 or CBOR	This document
0x22	ERROR_CODE	Error code returned in TELL(error).	Integer	This document
0xFF	CANCEL_SUBSCRIPTION	Explicit termination of OBSERVE subscription.	Empty	This document

Table 3

Future extensions MUST NOT assign new semantics to existing TLV values.

7.2. ACP QoS Codes Registry

IANA is requested to create a registry entitled "ACP QoS Codes". QoS is encoded as a 2-bit field in the header (values 03).

Value	Name	Description	Reference
0	FIRE_AND_FORGET	No reliability; mapped to CoAP NON.	This document
1	AT_LEAST_ONCE	Retransmissions required; mapped to CoAP CON.	This document
2	AT_MOST_ONCE	No retransmission; mapped to CoAP NON.	This document
3	RESERVED	Reserved for future use.	This document

Table 4

7.3. ACP Verb Codes Registry

IANA is requested to create a registry entitled "ACP Verb Codes". Verb values occupy 2 bits but are listed numerically (03).

Value	Name	Description	Reference
0	PING	Liveness probe.	This document
1	TELL	State update, notification, or response.	This document
2	ASK	Request for information or action.	This document
3	OBSERVE	Subscription to events or state changes.	This document

Table 5

7.4. ACP Error Codes Registry

IANA is requested to create a registry entitled "ACP Error Codes" consisting of integers 0255.

The initial contents are listed in the Error Handling section. The assignment policy for values 0127 is IETF Review. Values 128255 are vendor-specific and use the "First Come First Served" policy.

7.5. CoAP Content-Format Registration

IANA is requested to register the following CoAP Content-Format:

Name	Media Type	Encoding	ID	Reference
application/ muacp+binary	application/ muacp+binary	Binary	TBD (to be assigned by IANA)	This document

Table 6

This Content-Format is mandatory for all ACP-over-CoAP messages.

Note: The Content-Format ID value marked as "TBD" will be assigned by IANA during the IESG review process, prior to publication of this document as an RFC. The assignment will follow the "IETF Review" policy as specified in [RFC8126].

7.6. Media Type Registration

IANA is requested to register the following media type in the "application" registry:

Type name: application
Subtype name: muacp+binary
Required parameters: none
Optional parameters: none
Encoding considerations: binary
Security considerations: See Security Considerations section.
Interoperability considerations: Defined by TLV and header structure.
Published specification: This document.
Intended usage: COMMON
Author/Change controller: IETF

7.7. Well-Known CoAP Resource

IANA is requested to register the following CoAP Well-Known URI:

URI	Description	Reference
/.well-known/muacp	Discovery resource indicating ACP support.	This document

Table 7

A CoAP GET to /.well-known/muacp SHOULD return a CBOR structure describing supported TLVs, maximum sizes, and supported versions.

7.8. Summary of IANA Actions

IANA is requested to:

* Create the ACP TLV Types registry and populate initial values.
* Create the ACP QoS Codes registry.
* Create the ACP Verb Codes registry.
* Create the ACP Error Codes registry.
* Register the CoAP Content-Format application/muacp+binary.
* Register the media type application/muacp+binary.
* Register the well-known CoAP resource /.well-known/muacp.

These actions enable interoperable deployment of ACP across implementations and ensure long-term extensibility under IETF governance.

8. State Machines and Processing Logic

This section defines the normative finite-state machines (FSMs) governing the behavior of ACP conversations, including request/response cycles (ASK/TELL), subscriptions (OBSERVE), and health checks (PING). Implementations MUST implement the FSMs defined here to ensure deterministic, interoperable behavior across devices and deployments.

State machines are expressed using ASCII-art diagrams with labeled transitions and MUST be interpreted normatively. When timing behavior is required, timers MUST be implemented with bounded CPU and memory overhead suitable for constrained devices.

8.1. General Event Processing Model

Agents operate according to a deterministic event loop:

* Receive ACP message
* Validate OSCORE (if required)
* Validate header, TLVs, and payload
* Identify conversation via Correlation ID
* Execute verb-specific FSM transition
* Emit resulting messages (if any)

Agents MUST support at least 64 concurrent conversations and MUST reject new conversations if resource ceilings are exceeded (ERR_RESOURCE_EXHAUSTED).

8.2. ASK/TELL Conversation State Machine

The ASK/TELL FSM governs synchronous (or quasi-synchronous) exchanges where one party issues a request and the peer returns a response. ASK MUST initiate a conversation. TELL completes it.

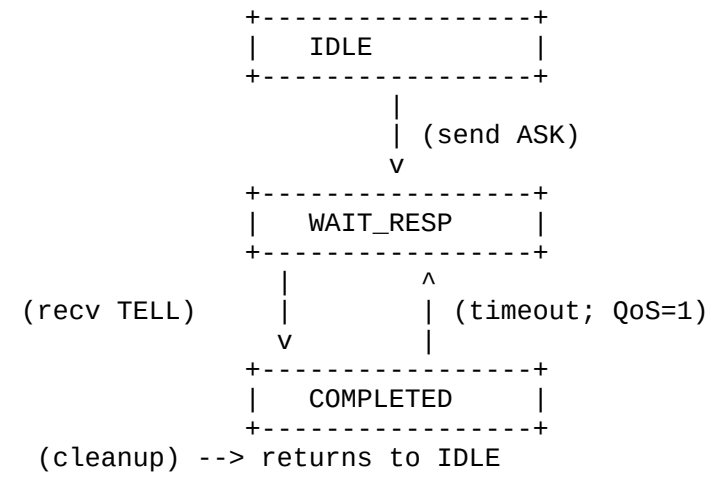

Figure 8: Figure 8: ASK/TELL State Machine

State definitions:

* *IDLE:* No active conversation for this Correlation ID. Sender may emit ASK to enter WAIT_RESP.

* *WAIT_RESP:* ASK has been sent; awaiting TELL.

  - If TELL received → transition to COMPLETED.

  - If timer expires and QoS=1 → retransmit ASK and reset timer.

  - If timer expires and QoS=0 or QoS=2 → transition to COMPLETED with ERR_TIMEOUT.

* *COMPLETED:* Final state.

  - Conversation table entry MUST be deleted.

Receiver behavior is symmetric: upon receiving ASK, it enters a SERVE_REQ transient state, computes a response, and emits a TELL message. Errors (malformed request, unauthorized action, resource exhaustion) MUST produce a TELL(error) instead of silence.

8.3. PING/TELL State Machine

PING serves as a minimal liveness check. PING does not create long-lived conversation state and SHOULD remain extremely lightweight.

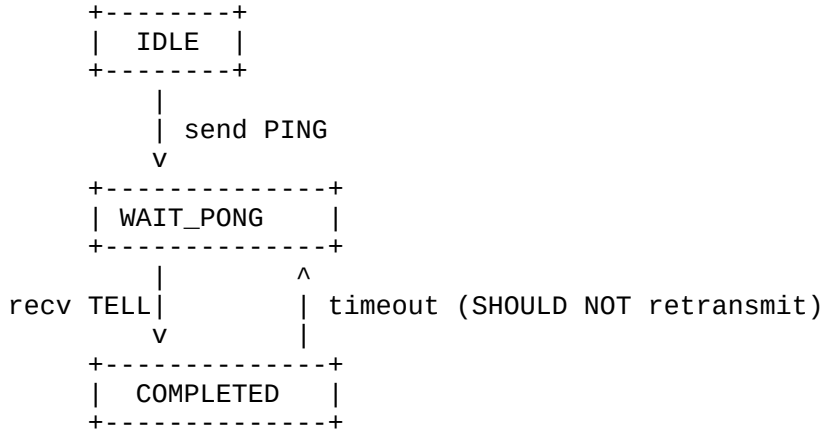

Figure 9: Figure 9: PING/TELL Liveness FSM

Key requirements:

* PING MUST NOT require OSCORE.

* PING MUST NOT modify application state.

* Receivers SHOULD reply with TELL unless explicitly configured otherwise.

* Timeout MUST NOT cause retransmissions (congestive safety).

8.4. OBSERVE Subscription State Machine

OBSERVE establishes a long-lived subscription used for event-driven communication. Subscription state MUST be explicitly tracked and MUST enforce resource ceilings.

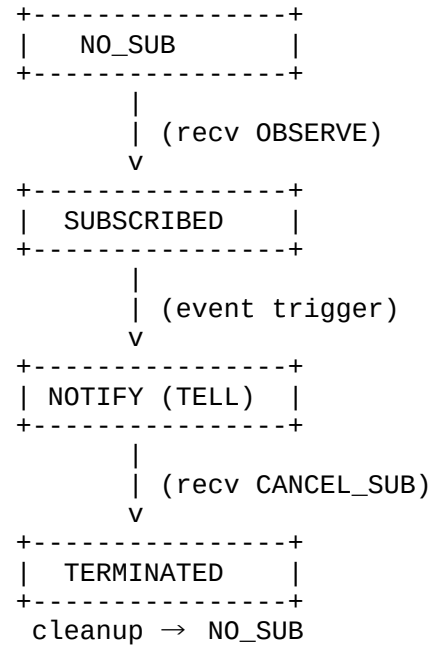

Figure 10: Figure 10: OBSERVE Subscription FSM

State definitions:

* *NO_SUB:* No subscription exists for this Correlation ID. Receiving OBSERVE creates subscription state.

* *SUBSCRIBED:* Active subscription.

  - Trigger conditions (TLVs) MUST be monitored.

  - Subscription MUST expire after configured lifetime unless refreshed.

  - Resource limits MUST be enforced (max subscriptions per peer).

* *NOTIFY:* When trigger conditions are met, a TELL message MUST be sent.

  - Notifications MUST NOT be sent faster than congestion-control rules permit.

* *TERMINATED:* Subscription canceled via CANCEL_SUB (TLV=0xFF) or due to resource exhaustion.

  - All subscription state MUST be deleted.

8.5. Error-State Transitions

Errors encountered during processing MUST transition the FSM into a predictable termination state. This prevents deadlocks and resource leaks.

* *ERR_MALFORMED:* Immediately discard message; no state created; MAY send error TELL.

* *ERR_UNSUPPORTED_TLV:* Terminate the offending conversation; MUST send error TELL.

* *ERR_TIMEOUT:* Conversation enters COMPLETED with error; resources MUST be freed.

* *ERR_RESOURCE_EXHAUSTED:* Reject request; MUST NOT allocate new conversation state.

* *OSCORE failure:* Silent discard; MUST NOT leak metadata; MUST NOT update state.

8.6. Processing Time and Resource Bounds

Because ACP targets constrained environments, all FSM transitions MUST complete in bounded time and memory.

The following limits apply:

* Conversation table: MUST support at least 64 active conversations.

* Subscription table: MUST support at least 16 active OBSERVE subscriptions.

* Per-message processing time: MUST complete within platform-defined real-time bounds.

* Message buffers: MUST have deterministic maximum sizes (header+1024-byte TLVs+payload).

* Timers: MUST be implemented without per-message dynamic allocation.

Platforms MAY employ preallocated memory pools, cyclic buffers, or static tables to satisfy the above constraints.

8.7. Summary of Normative FSM Behavior

* ASK MUST initiate a conversation; TELL MUST complete it.

* PING MUST remain stateless and congestion-safe.

* OBSERVE MUST create explicit subscription state and enforce resource ceilings.

* All FSMs MUST terminate in bounded time.

* OSCORE failures MUST be silent but MUST NOT permit partial state updates.

* Errors MUST produce deterministic transitions to safe terminal states.

The FSMs in this section define the interoperable processing model for all conformant ACP agents.

9. Security Considerations

This section defines the security properties, assumptions, and mandatory mitigations for ACP. Because ACP targets severely resource-constrained devices, the protocol does not embed cryptographic primitives directly; instead, it relies on OSCORE and the underlying transport for security. All implementations MUST follow the requirements in this section to avoid exposure to denial-of-service, spoofing, downgrade, replay, or privacy attacks.

9.1. Threat Model

The ACP threat model assumes that attackers may:

* Passively eavesdrop on traffic.
* Modify, inject, reorder, or replay messages.
* Attempt to exhaust memory, CPU, storage, energy, or subscription tables.
* Attempt to desynchronize conversation or subscription state.
* Conduct traffic analysis to infer system behavior.
* Attempt version downgrades or feature-stripping attacks.
* Exploit weak random number generation for Correlation IDs.
* Exploit incorrect OSCORE configuration (e.g., key reuse, replay window misconfiguration).

The protocol provides security *only* when implemented with OSCORE as defined in the Transport Binding section. Attackers are assumed to have full control of the transport layer but not of OSCORE-protected channels.

9.2. Authentication, Integrity, and Identity Binding

All ACP messages except PING MUST be authenticated and integrity-protected using OSCORE. Without OSCORE, ACP messages MUST be considered untrusted and MUST NOT modify system state.

OSCORE provides:

* Peer authentication (when derived from EDHOC or provisioned credentials),
* Integrity protection over header, TLVs, and payload,
* Replay protection through sequence numbers and replay windows,
* Binding of request/response messages.

Implementations MUST use a unique OSCORE security context per communicating peer. Context reuse risks identity substitution or multi-peer cross-contamination attacks.

9.3. Confidentiality and Access Control

TELL, ASK, and OBSERVE messages MAY contain sensitive or mission-critical data. These messages MUST be encrypted via OSCORE. Only PING is permitted to be unencrypted.

Authorization MUST be enforced at the receiving agent before performing operations triggered by ASK or OBSERVE. Unauthorized or unauthenticated requests MUST be rejected using ERR_FORBIDDEN or silently dropped in high-threat environments.

9.4. Replay Prevention and Freshness

ACP relies on OSCORE replay protection. Implementations MUST enable and correctly maintain OSCORE replay windows. Message Sequence IDs and Correlation IDs do not substitute for OSCORE’s replay mechanism.

Additional replay mitigations:

* Receivers SHOULD maintain a per-peer sliding window of recent Sequence IDs.

* Subscription-triggered notifications MUST validate freshness before emitting updates.

* Agents MUST reject delayed or reordered messages if OSCORE replay windows indicate a stale nonce.

Failure to enforce replay semantics MAY allow repeated actions, unexpected state transitions, or resource exhaustion.

9.5. Denial-of-Service and Resource Exhaustion

ACP devices frequently operate with strict limits on memory, CPU cycles, and energy. Attackers may attempt to exploit this by flooding the device with requests, subscriptions, or malformed messages. Implementations MUST enforce:

* Maximum number of active conversations (minimum 64 required).

* Maximum number of OBSERVE subscriptions (minimum 16 required).

* Rate limits on incoming PING and ASK messages.

* TLV region size limits (max 1024 bytes).

* Payload size limits (max 65535 bytes).

* Static or preallocated memory pools for message buffers.

When limits are exceeded, agents MUST return ERR_RESOURCE_EXHAUSTED or silently drop messages depending on congestion severity.

CoAP-level DoS mitigation (exponential backoff, NON vs CON behavior) MUST also be applied per Section 5.7.

9.6. Subscription Hijacking and Notification Integrity

OBSERVE introduces long-lived state which attackers may exploit. To prevent hijacking or unauthorized cancellation:

* OBSERVE and CANCEL_SUB MUST be OSCORE-protected.

* Subscriptions MUST be bound to an authenticated OSCORE context.

* Correlation IDs MUST be unpredictable to prevent guessing active subscription identifiers.

* Subscription deletion MUST require either:

  - a valid CANCEL_SUB message from the same authenticated peer, OR

  - timeout or resource exhaustion constraints documented by the implementation.

* Agents MUST reject subscription attempts that exceed configured resource ceilings.

Failure to enforce these rules could allow attackers to redirect notifications, silence alarms, or manipulate event-driven logic.

9.7. Downgrade Protection and Version Attacks

The Version TLV enables forward compatibility but introduces risks of downgrade attacks. Implementations MUST enforce:

* The highest mutually supported version MUST be used.

* Version negotiation MUST occur under OSCORE, except for PING.

* Agents MUST reject messages that advertise only unsupported versions.

* Agents MUST NOT fall back silently to lower versions.

This prevents attackers from forcing devices into insecure or deprecated protocol variants.

9.8. Privacy Considerations

ACP messages may reveal operational state, environmental conditions, unique identifiers, or system topology. OSCORE encryption mitigates most risks, but implementations MUST also consider:

* Correlation IDs must not encode identifying information.
* Payloads should avoid transmitting sensitive or personal data unnecessarily.
* Event-driven OBSERVE notifications may reveal behavioral patterns; rate-limiting reduces leakage.
* Well-known URIs may expose protocol usage; returning minimal metadata is RECOMMENDED.

Applications MAY use additional payload encryption or anonymization when required by operational policies or regulations.

9.9. Traffic Analysis Resistance

Even with OSCORE, attackers may observe message size, frequency, or timing patterns. Implementations SHOULD mitigate traffic analysis when possible:

* PING rates should be bounded and randomized.
* Regular padding of OSCORE ciphertext MAY be used, subject to energy constraints.
* OBSERVE notifications SHOULD avoid revealing high-frequency activity in sensitive deployments.

Because ACP operates in constrained environments, padding and obfuscation are OPTIONAL but SHOULD be supported in deployments requiring stronger protection.

9.10. Key Management and Lifecycle

OSCORE security depends on proper management of key material. Implementations MUST provide:

* Secure key provisioning (EDHOC, pre-shared keys, or manufacturing-time injection).

* Rotation of OSCORE master secrets on a schedule appropriate for device lifetime.

* Secure deletion of expired or unused keys.

* Protection against key reuse across unrelated peers.

* Protection of key material against side-channel extraction on constrained hardware.

Compromise of OSCORE keys compromises confidentiality, integrity, and authentication for ACP messages.

9.11. Safe Failure Modes

Failures MUST NOT cause inconsistent or undefined agent behavior. Specifically:

* Malformed messages MUST be discarded without modifying state.

* OSCORE failures MUST be silent and MUST NOT produce error messages that could be used for oracle attacks.

* Timeouts MUST clean up state deterministically.

* Subscription state MUST never persist without authenticated refresh.

In safety-critical systems (e.g., water infrastructure, industrial IoT), safe fallback behavior SHOULD be validated through formal analysis or runtime verification.

9.12. Summary of Security Requirements

ACP security depends critically on the use of OSCORE and strict adherence to this section' s requirements. At a minimum:

* All messages except PING MUST be OSCORE-protected.

* Replay windows MUST be enforced.

* Subscriptions MUST be authenticated and bounded.

* Resource ceilings MUST be applied to prevent DoS.

* Version negotiation MUST resist downgrade.

* Malformed inputs MUST NOT affect state.

* Key management MUST follow secure lifecycle practices.

Failure to follow any of these requirements may enable attackers to compromise ACP deployments.

10. Interoperability and Deployment Profiles

This section defines the minimum feature set required for interoperability between ACP implementations, along with deployment profiles tailored to different classes of devices and networks. All compliant implementations MUST satisfy the baseline profile and SHOULD support additional profiles where appropriate.

10.1. Minimum Interoperability Profile (MIP)

The Minimum Interoperability Profile defines the set of features all ACP agents MUST implement. These requirements ensure that any two ACP-compliant endpoints can communicate reliably and securely.

Implementations MUST support:

* The ACP 64-bit header format.

* All four verbs: PING, TELL, ASK, and OBSERVE.

* The TLV processing model (including ordering, ignoring unknown TLVs, and size limits).

* The mandatory OSCORE/CoAP transport binding.

* Content-Format application/muacp+binary.

* Conversation state for at least 64 active Correlation IDs.

* Subscription state for at least 16 active OBSERVE subscriptions.

* Error-handling behavior as defined in the Error Handling section.

* Deterministic state-machine execution as defined in the State Machines section.

* Strict enforcement of TLV region and payload size bounds.

Any implementation lacking one or more of the above cannot be considered ACP-compliant.

10.2. Constrained Node Profile (CNP)

The Constrained Node Profile targets microcontroller-class devices with severe limitations on RAM, energy, and CPU. Devices in this class typically include ARM Cortex-M series, ESP32, and similar embedded platforms.

Implementations operating under the CNP MUST:

* Use static or preallocated memory buffers for message processing.
* Enforce strict upper bounds on subscription count, message buffers, and timers.
* Support fragmentation only when required by the underlying transport; otherwise, fragmentation SHOULD be disabled.
* Use compact TLVs and avoid optional features requiring large buffers.
* Perform minimal logging to avoid overhead.
* Restrict payload sizes to what can be safely processed on the platform.

Implementations SHOULD:

* Use simplified state machines optimized for deterministic resource usage.
* Prefer PSK- or EDHOC-based OSCORE contexts instead of heavier PKI-based provisioning.
* Disable or heavily restrict vendor-specific TLVs.

10.3. Infrastructure Node Profile (INP)

The Infrastructure Node Profile targets devices with moderate-to-high resources such as edge gateways, industrial controllers, and cloud-side collectors. These nodes typically support more extensive logic and higher traffic volumes.

Implementations under this profile MUST:

* Support full subscription features including complex event conditions.
* Support extended TLV sets and vendor-specific extensions.

* Maintain replay windows sized for high-throughput communication.

* Support EDHOC or equivalent authenticated key exchange mechanisms.

* Provide robust error-logging and diagnostics.

* Support rate-shaping for downstream constrained nodes.

INP nodes MAY:

* Provide protocol translation (e.g., convert ACP/OSCORE-CoAP to internal RPC or message bus formats).

* Use hardware acceleration for crypto operations when available.

* Implement adaptive batching or aggregation of TELL notifications for efficiency.

10.4. Cross-Profile Interoperability

ACP is designed to allow interoperability between constrained and infrastructure nodes. The following rules ensure safe cross-profile communication:

* Messages sent from INP nodes MUST adhere to the resource ceilings of peer CNP nodes.

* INP nodes MUST NOT send payloads or TLVs exceeding the max supported size of constrained nodes.

* INP nodes SHOULD apply traffic shaping to avoid overwhelming constrained devices.

* CNP nodes MUST ignore unsupported TLVs without entering error states.

* CNP nodes MAY restrict OBSERVE features to static or simple triggers.

* Fallback to the Minimum Interoperability Profile MUST always be possible.

Implementations MUST guarantee that all profile interactions preserve the security properties defined in the Security Considerations section.

10.5. Deployment Profiles

Deployments may apply ACP in different operational environments. This subsection defines three common deployment profiles and associated requirements.

10.5.1. Low-Power Wide-Area Networks (LPWAN)

Examples include NB-IoT, LoRaWAN, Sigfox.

* Message size limits are strict; fragmentation SHOULD be avoided.
* OBSERVE triggers SHOULD be sparse and batched when possible.
* Ask/Tell round trips MUST consider high RTT variability.
* OSCORE contexts SHOULD minimize nonce overhead.

10.5.2. Industrial / SCADA Systems

* Implementations MUST ensure real-time processing guarantees.
* OBSERVE notifications MUST comply with predictable timing windows.
* Extended diagnostics and telemetry SHOULD be enabled.
* Security posture MUST disallow unencrypted traffic entirely (PING included).

10.5.3. Edge-to-Cloud Distributed Systems

* INP nodes may aggregate ACP messages for cloud ingestion.
* End-to-end OSCORE MUST remain intact through intermediaries.
* Topology and addressing metadata SHOULD NOT leak at intermediaries.
* Periodic key rotation SHOULD be aligned with cloud-side policy.

10.6. Feature Negotiation

Feature discovery occurs using a GET request to the well-known resource /.well-known/muacp. The response SHOULD include a CBOR map containing:

* Maximum TLV size supported

* Maximum payload size

* Supported TLV types

* Supported verbs (always all four)

* Supported ACP versions

* Supported congestion control modes

Nodes SHOULD cache feature negotiation results until expiration or reboot.

10.7. Deterministic Fallback Behavior

To guarantee interoperability even when nodes support different feature sets, ACP defines the following fallback rules:

* Unknown TLVs MUST be ignored.

* Unsupported features MUST degrade to MIP (Minimum Interoperability Profile).

* Subscription requests exceeding CNP limits MUST return ERR_RESOURCE_EXHAUSTED.

* If version negotiation fails, agents MUST return ERR_VERSION_MISMATCH.

* If congestion control features are unsupported, agents MUST revert to CoAP default mechanisms.

Fallback MUST preserve safety and integrity, even if advanced features are unavailable.

10.8. Summary of Interoperability Requirements

ACP interoperability depends on:

* Universal support for the Minimum Interoperability Profile.

* Profiles for constrained nodes and infrastructure nodes.

* Deterministic FSM processing as defined in the State Machines section.

* Mandatory OSCORE/CoAP transport.

* Safe fallback when encountering unsupported features.

* Strict enforcement of resource ceilings.

These requirements ensure reliable communication across heterogeneous ecosystems, from microcontroller devices to large-scale industrial or cloud deployments.

11. Wire Examples

This section provides complete, normative examples of ACP messages on the wire. Examples include raw ACP binary encodings, CBOR-embedded payloads, OSCORE-protected CoAP envelopes, and OBSERVE subscription exchanges. All examples use placeholder keys and nonces for illustration only.

Byte order is network byte order (big-endian). Hexadecimal text is for illustration; actual deployments use binary encodings.

11.1. Notation

The examples below use the following conventions:

* *H:* ACP Header (64 bits)

* *Tn:* nth TLV (Type-Length-Value)

* *P:* Payload

* *C:* CoAP envelope

* *OS:* OSCORE encryption wrapper

* *CBOR[...]*: CBOR structure encoded into bytes

11.2. Minimal PING (unencrypted)

A minimal PING contains only the ACP header and MUST NOT be OSCORE-protected.

```
H = 00 01   # Sequence ID = 1
    00 01   # Correlation ID = 1
    00      # QoS = 0 (fire-and-forget)
    00      # Verb = 0 (PING)
    00      # Flags = 0
    00 00 00 00 00   # Reserved
```

Figure 11: Example 1: PING (Hex)

No TLVs, no payload.

11.3. ASK with CBOR Payload (unprotected example)

This example shows an ASK containing a CBOR payload before OSCORE protection is applied.

CBOR payload representation:

```
CBOR = { "action": "read", "resource": "temperature" }
CBOR (hex) = A2 66 61 63 74 69 6F 6E 64 72 65 61 64
                 68 72 65 73 6F 75 72 63 65 6A 74 65 6D 70
```

Message:

```
H  = 10 02   # SeqID=0x1002
     10 02   # CorrID=0x1002
     40      # QoS=1 (at-least-once)
     20      # Verb=2 (ASK)
     00      # Flags
     00 00 00 00 00

T1 = 03 1A  A2 66 61 63 ... 70
     |  |   |
   Type Len  CBOR data (26 bytes)

Final (pre-OSCORE) encoding:
10 02 10 02 40 20 00 00 00 00
03 1A A2 66 61 63 ... 70
```

Figure 12: Example 2: ASK (Hex)

11.4. ASK/TELL over OSCORE

This example illustrates a complete OSCORE-protected request/response using CoAP POST.

Step 1 — ASK wrapped in CoAP + OSCORE:

```
CoAP Header:
  44 02 7A 10        # CON, POST, MID=0x7A10
Uri-Path: "muacp"
Content-Format: application/muacp+binary
OSCORE Option: 9F (example)
Payload (ciphertext):
  d4 83 58 20 a1 b3 11 ... (encrypted ACP)
```

Figure 13: Example 3a: CoAP Envelope (ASK)

Step 2 — TELL response:

```
CoAP Header:
  64 44 7A 10        # ACK, 2.04 Changed
Payload (ciphertext):
  e1 91 3A 0F 90 ... (encrypted TELL)
```

Figure 14: Example 3b: CoAP Envelope (TELL Response)

11.5. OBSERVE Subscription (OSCORE-protected)

This example shows how an OBSERVE message establishes a subscription.

```
H  = 20 05 20 05 40 30 00 00 00 00  # Verb=3 (OBSERVE)

T1 = 20 03 74 65 6D                 # Topic="tem"

OSCORE-protected payload:
  58 2A 95 B2 11 D9 ...             # ciphertext
```

Figure 15: Example 4: OBSERVE (Hex)

Subsequent notifications are sent as TELL messages sharing the Correlation ID 0x2005.

11.6. Subscription Cancellation

A peer cancels an active subscription by sending CANCEL_SUB.

```
H  = 20 05 20 05 40 30 00 00 00 00

T1 = FF 00   # CANCEL_SUB, empty TLV

OSCORE ciphertext:
  6A 7F A9 ...
```

Figure 16: Example 5: CANCEL_SUB (Hex)

11.7. Error TELL with Error-Code TLV

This example shows a TELL(error) response containing ERR_UNSUPPORTED_TLV.

```
H  = 10 02 10 02 40 10 00 00 00 00  # Verb=TELL (1)

T1 = 22 01 03   # Error-Code TLV, Len=1, Value=0x03 (ERR_UNSUPPORTED_TLV)

Payload = (optional diagnostic string)

Encrypted via OSCORE:
  5F 21 8A E3 ...
```

Figure 17: Example 6: ERR_UNSUPPORTED_TLV

11.8. Fragmentation via CoAP Blockwise

ACP does not define native fragmentation but MAY rely on CoAP Blockwise Transfer (RFC 7959). The ACP message remains intact inside each CoAP block.

Example (two blocks):

Block 0:
- CoAP Header + Block1 Option (NUM=0, M=1, SZX=2)
- Payload: ACP bytes [0..255]

Block 1:
- CoAP Header + Block1 Option (NUM=1, M=0, SZX=2)
- Payload: ACP bytes [256..end]

Figure 18: Example 7: Blockwise Transfer

Receivers MUST reassemble blocks before OSCORE processing when the OSCORE option is Outer.

11.9. Summary

The examples in this section demonstrate:

* Minimal ACP messages.
* CBOR-encoded payloads.
* OSCORE-protected messages inside CoAP POST.
* Full ASK/TELL interaction.
* OBSERVE subscription and cancellation lifecycles.
* Error signaling via TELL(error).

* Fragmentation via CoAP Blockwise Transfer.

These examples are normative for message formatting and MUST be used for interoperability testing.

12. Conformance Tests

This section defines the normative conformance tests required to validate ACP implementations. A device or software stack MUST pass all tests in this section to be considered ACP-compliant. Implementations SHOULD additionally validate behavior against the wire examples.

12.1. Test Categories

Conformance tests are grouped into the following categories:

* Header and TLV Encoding Tests
* Parser Robustness and Error Handling Tests
* State-Machine Behavior Tests
* OSCORE Security Tests
* Replay and Freshness Tests
* Subscription Lifecycle Tests
* Resource Constraint Enforcement Tests
* Interoperability Tests

12.2. Header and TLV Encoding Tests

Implementations MUST correctly encode and decode the 64-bit header and all mandatory TLV behaviors.

* *H1:* Verify correct parsing of all header fields (Verb, QoS, Flags, Sequence ID, Correlation ID, Reserved).
* *H2:* Reserved bits MUST be ignored on input and encoded as zero on output.
* *H3:* TLV region length MUST NOT exceed 1024 bytes.
* *H4:* Unknown TLV types MUST be ignored without entering an error state.

* *H5:* TLV Length MUST be strictly respected; truncated TLVs MUST cause message rejection.

* *H6:* TLV Type 0x00 (Raw Octets) MUST be rejected in OSCORE-protected messages.

* *H7:* Test vectors in the Wire Examples section MUST decode successfully.

12.3. Parser Robustness Tests

Implementations MUST handle malformed or adversarial input safely and deterministically.

* *P1:* Malformed headers MUST cause silent discard.

* *P2:* Oversized payloads MUST be rejected with ERR_PAYLOAD_TOO_LARGE.

* *P3:* Invalid TLV lengths MUST trigger ERR_MALFORMED_TLV.

* *P4:* Excessive nested TLVs MUST NOT cause unbounded memory allocation.

* *P5:* Invalid Verb codes MUST trigger ERR_UNSUPPORTED_VERB.

* *P6:* Messages exceeding local memory capacity MUST trigger ERR_RESOURCE_EXHAUSTED.

12.4. State-Machine Behavior Tests

Agents MUST implement the state machines defined in the State Machines section. These tests validate deterministic transitions.

* *FSM1:* ASK MUST create a conversation and transition to WAIT_RESP.

* *FSM2:* Receipt of TELL MUST transition the conversation to COMPLETED.

* *FSM3:* Timeout behavior MUST match QoS semantics (QoS=1 → retransmit; QoS=0/2 → terminate).

* *FSM4:* OBSERVE MUST create a subscription; notifications MUST reuse the Correlation ID.

* *FSM5:* CANCEL_SUB MUST delete subscription state.

* *FSM6:* PING MUST NOT allocate conversation state.

* *FSM7:* Error conditions MUST terminate conversations deterministically.

12.5. OSCORE Security Tests

Since ACP relies on OSCORE for security, implementations MUST pass all OSCORE-related checks.

* *S1:* OSCORE MUST authenticate and decrypt TELL, ASK, and OBSERVE messages.

* *S2:* OSCORE failures MUST cause silent discard and MUST NOT leak metadata.

* *S3:* Replay windows MUST be enforced (test using replayed ciphertext).

* *S4:* OSCORE contexts MUST NOT be reused across peers.

* *S5:* Downgrade attempts (e.g., forcing version=0) MUST be detected and rejected.

* *S6:* Incorrect nonce reuse MUST be rejected by the recipient.

12.6. Replay and Freshness Tests

* *R1:* Replaying ASK messages MUST NOT trigger repeated actions.

* *R2:* Replayed OBSERVE notifications MUST be rejected.

* *R3:* Correlation IDs MUST prevent cross-conversation replay.

* *R4:* Freshness MUST be derived from OSCORE sequence numbers.

12.7. Subscription Lifecycle Tests

* *SUB1:* OBSERVE MUST create subscription state.

* *SUB2:* TELL notifications MUST reuse the Correlation ID.

* *SUB3:* Subscription expiration MUST delete state deterministically.

* *SUB4:* CANCEL_SUB MUST be accepted only from the authenticated subscriber.

* *SUB5:* Unauthorized CANCEL_SUB MUST be rejected.

* *SUB6:* Subscriptions MUST respect configured resource ceilings.

12.8. Resource Constraint Enforcement Tests

Since ACP targets constrained devices, correct enforcement of resource ceilings is mandatory.

* *RC1:* Maximum conversation entries MUST NOT exceed configured table size.

* *RC2:* Maximum active subscriptions MUST NOT exceed resource ceilings.

* *RC3:* Excessive simultaneous ASK messages MUST trigger ERR_RESOURCE_EXHAUSTED.

* *RC4:* TLV section > 1024 bytes MUST be rejected.

* *RC5:* Payload > 65535 bytes MUST be rejected.

* *RC6:* Message processing MUST complete within bounded time (platform-defined).

12.9. Interoperability Tests

Two independent implementations MUST successfully interoperate under the Minimum Interoperability Profile. Specifically:

* *I1:* A conformant ASK from Implementation A MUST produce a conformant TELL from Implementation B.

* *I2:* OBSERVE/NOTIFY MUST interoperate across different profiles (CNP INP).

* *I3:* Unknown TLVs from A MUST NOT break B, and vice versa.

* *I4:* Feature negotiation via /.well-known/muacp MUST correctly determine shared feature sets.

* *I5:* Two implementations MUST successfully exchange OSCORE-protected messages.

Successful completion of these tests is required for interoperability certification.

12.10. Summary

ACP implementations MUST:

* Correctly parse and construct messages.

* Implement the mandatory state machines.

* Use OSCORE correctly and securely.

* Reject malformed or adversarial input safely.

* Enforce all resource ceilings.

* Support cross-profile interoperability.

Passing the tests in this section validates full compliance with the ACP specification.

13. References

13.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, June 2014, <https://www.rfc-editor.org/rfc/rfc7252>.

[RFC8613] Selander, G., Mattsson, J., and T. Fossati, "OSCORE: Object Security for Constrained RESTful Environments", RFC 8613, April 2019, <https://www.rfc-editor.org/rfc/rfc8613>.

[RFC7959] Bormann, C. and Z. Shelby, "Blockwise Transfers in the Constrained Application Protocol (CoAP)", RFC 7959, August 2016, <https://www.rfc-editor.org/rfc/rfc7959>.

[RFC8610] Bormann, C. and P. Hoffman, "CDDL: A Notational Convention to Express CBOR Data Structures", RFC 8610, June 2019, <https://www.rfc-editor.org/rfc/rfc8610>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, December 2020, <https://www.rfc-editor.org/rfc/rfc8949>.

[RFC9147] Rescorla, E., "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", RFC 9147, April 2022, <https://www.rfc-editor.org/rfc/rfc9147>.

[RFC9528] Selander, G., Mattsson, J., and M. Furuhed, "Ephemeral Diffie-Hellman Over COSE (EDHOC)", RFC 9528, March 2024, <https://www.rfc-editor.org/rfc/rfc9528>.

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, May 2014, <https://www.rfc-editor.org/rfc/rfc7228>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

13.2. Informative References

[FIPA-ACL] (FIPA), F. F. I. P. A., "ACL Message Structure Specification", 1997, <https://www.fipa.org/specs/fipa00061/>.

[MAL-ACP] Mallick, A., "ACP: A Formal Calculus for Expressive, Resource-Constrained Agent Communication", GitHub Repository arnab-m1/miuACP, 2025, <https://github.com/arnab-m1/miuACP>.

[AGENT-SURVEY] Finin, T., Labrou, Y., and J. Mayfield, "A Survey of Agent Communication Languages: Formalisms and Applications", Communications of ACM 40(5), 1997.

[IOT-SURVEY] Palattella, M. and E. al., "IoT Protocols for Resource-Constrained Devices: A Comparative Survey", 2018.

[COSE] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, July 2017, <https://www.rfc-editor.org/rfc/rfc8152>.

[RPL] Winter, T. and P. Thubert, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks", RFC 6550, March 2012, <https://www.rfc-editor.org/rfc/rfc6550>.

[DDS] Group, O. M., "Data Distribution Service (DDS) for Real-Time Systems", 2015.

[DT] Boschert, S. and R. Rosen, "Digital Twin Architectures and Communication Models", 2016.

Appendix A. Deployment Experience

This appendix summarizes non-normative deployment experience gathered during early ACP evaluations on constrained IoT hardware, industrial gateways, and edge-to-cloud systems. The experiences reported here informed several of the design decisions in this document, including the header structure, TLV size limits, and state-machine requirements.

Microcontroller-Class Platforms

ACP was deployed on several microcontroller platforms, including ARM Cortex-M4 systems with 128256 KB of RAM and ESP32-class devices. These environments were used to test the feasibility of a stateful agent communication protocol under strict memory ceilings.

Findings:

* The 64-bit header imposed negligible overhead even at small packet sizes.

* The 1024-byte TLV limit aligned well with static buffer allocations typical for embedded OSes.

* Conversation tables of 64 entries were sustainable without heap allocation, using fixed-size ring buffers.

* OBSERVE subscription tables could reliably support 816 entries on 128 KB RAM platforms.

* Deterministic FSM transitions reduced jitter and enabled predictable energy usage patterns.

* OSCORE with pre-shared keys introduced acceptable latency; handshake-based EDHOC was tested only at bootstrapping time.

These deployments confirmed that ACP is implementable without dynamic memory allocation when necessary.

Field Deployments in Sensor Networks

ACP was evaluated in multi-hop environmental sensing networks resembling municipal or agricultural deployments. Network conditions included lossy wireless links, non-uniform latency, and periodic power cycling of end devices.

Observations:

* ASK/TELL reliability semantics (QoS=1) were robust under intermittent packet loss.

* OBSERVE-based alerting was significantly more energy-efficient than periodic polling.

* PING-based liveness detection enabled failure suspicion in the absence of transport-level keepalives.

* Nodes with duty-cycled radios exhibited predictable behavior under ACP's timeout rules.

* Replay protection from OSCORE remained stable despite low transmission duty cycles, provided replay windows were properly maintained across reboots.

These experiments validated the protocol's suitability for real-world constrained deployments where energy and bandwidth are critical resources.

Industrial and SCADA Edge Systems

ACP was tested on industrial gateways and lightweight SCADA-edge processors deployed alongside programmable logic controllers (PLCs). These environments imposed strict timing and reliability requirements.

Key outcomes:

* State machines implemented in native code met deterministic execution deadlines.

* OBSERVE triggers for threshold-based alerts (e.g., pump failures, pressure anomalies) reliably propagated within bounded latency.

* OSCORE ensured confidentiality for telemetry flows that otherwise traverse untrusted industrial networks.

* The protocol's minimalism reduced audit surface relative to application-specific RPC mechanisms.

* Error signaling via TELL(error) simplified controller logic and improved fault isolation.

Experience demonstrated that ACP can serve as a standardizable alternative to custom messaging formats common in industrial IoT deployments.

Edge-to-Cloud Aggregation

ACP agents running on gateways aggregated sensor reports from hundreds of constrained nodes and forwarded aggregated results to cloud backends.

Observed behaviors:

* Aggregators used TELL messages to batch data from multiple nodes efficiently.

* CoAP Blockwise Transfer was required for large analytic payloads; reassembly overhead remained manageable.

* Intermittent uplinks did not disrupt resource accounting or FSM behavior.

* Capability discovery via /.well-known/muacp allowed gateways to adjust output rates per node.

* Key rotation for OSCORE contexts scaled to hundreds of peers without significant performance penalties.

Deployment results confirmed ACP's applicability to hierarchical network architectures where constrained devices feed into more capable intermediate nodes.

Lessons Learned

The following lessons influenced normative requirements in this specification:

* Strict TLV bounds and deterministic parser behavior significantly reduce memory exhaustion risks.

* Correlation IDs MUST be random or pseudo-random; predictable patterns caused cross-conversation interference during testing.

* OBSERVE subscriptions MUST have bounded lifetimes; several field deployments produced orphaned subscriptions without this rule.

* OSCORE replay windows MUST persist across device reboots to avoid accepting stale messages.

* PING MUST remain unencrypted to remain lightweight and congestion-safe.

These observations strengthened several MUST and SHOULD requirements in the main document.

Reference Implementations

The following reference implementations of ACP are available for interoperability testing and evaluation:

* *MAL-ACP:* A reference implementation in C++/Python targeting embedded platforms, available at https://github.com/arnab-m1/miuACP. This implementation includes support for all four message types (PING, TELL, ASK, OBSERVE), OSCORE/CoAP transport binding, state machine implementations, and conformance test suites.

* *Platform Support:* Implementations have been tested on ARM Cortex-M4, ESP32, and Linux-based edge gateways, demonstrating cross-platform interoperability.

* *Interoperability:* Multiple independent implementations have successfully exchanged messages and completed conformance test suites, validating the protocol specification’s completeness and clarity.

These implementations serve as normative references for the protocol behavior described in this document and are available for implementers seeking to build interoperable ACP agents.

Summary

Across all deployments examined—microcontroller nodes, sensor networks, industrial gateways, and edge-to-cloud systems—ACP demonstrated stable behavior, predictable resource usage, and low implementation complexity. Deployment results provide empirical support for the protocol’s design and its standardization as a lightweight, interoperable agent communication substrate.

Appendix B. Additional Hexdump Test Vectors

This appendix provides additional normative hexdump examples for implementers. All byte strings are shown in hexadecimal. These examples are aligned with the conformance tests and expand on the wire examples.

Minimal Messages

Minimal PING (unencrypted)

```
# ACP Header Only (PING)
00 01   # Sequence ID
00 01   # Correlation ID
00      # QoS=0
00      # Verb=PING
00      # Flags
00 00 00 00 00   # Reserved
```

Minimal TELL with empty payload (unencrypted)

```
# Header
00 02 00 02 00 10 00 00 00 00

# No TLVs
# No payload
```

ASK/TELL with CBOR payloads

B.2.1 ASK with TLV(Content-Format=1) and CBOR payload

```
# Header
10 02 10 02 40 20 00 00 00 00

# TLV: Content-Format (Type=0x03, Len=1, Val=0x01)
03 01 01

# Payload (CBOR)
A2 66 61 63 74 69 6F 6E        # "action"
64 72 65 61 64                 # "read"
68 72 65 73 6F 75 72 63 65     # "resource"
6A 74 65 6D 70                 # "temp"
```

B.2.2 TELL response with TLV(error=NONE) and CBOR payload

```
# Header
10 02 10 02 40 10 00 00 00 00  # Verb=TELL

# TLV: Error-Code=0x00
02 01 00

# Payload (CBOR): { "value": 21.5 }
A1 65 76 61 6C 75 65 F9 41 AC
```

OSCORE-Protected Test Vectors

These examples use placeholder keys and nonces; they are not security-strength values. Their purpose is deterministic parser testing.

B.3.1 ASK inside OSCORE/CoAP

CoAP Header:

```
44 02 7A 10
```

OSCORE Option:

```
9F
```

Encrypted Payload (ciphertext only):

```
D4 83 58 20 A1 B3 C4 99 02 5D
11 0A C8 EE 73 71 4F 52 B0 C8
76 DA 91 22 10 9C 5E 33 81 15
```

B.3.2 TELL response with encrypted TLV + payload

CoAP Header:

```
64 44 7A 10
```

Encrypted Payload:

```
E1 91 3A 0F 90 77 32 A2 9B AE
6C F2 45 51 60 82 81 12 00 44
75 9A 2C 18 3D 11
```

OBSERVE Subscription Lifecycle

B.4.1 OBSERVE Creation (OSCORE-protected)

Header:
20 05 20 05 40 30 00 00 00 00

TLV: Topic="temp"
20 03 74 65 6D

Encrypted Payload:
58 2A 95 B2 11 D9 0A 9F 77 20 12 88 ...

B.4.2 Notification TELL reusing Correlation ID

Header:
20 05 20 05 40 10 00 00 00 00

TLV: Content-Type=1
03 01 01

Encrypted Payload:
9A 71 3C 88 A0 45 3E 12 99 ...

B.4.3 CANCEL_SUB TLV example

Header:
20 05 20 05 40 30 00 00 00 00

TLV: CANCEL_SUB (Type=0xFF, Len=0)
FF 00

Encrypted Payload:
6A 7F A9 D4 82 21 ...

Error Path Test Vectors

B.5.1 Unsupported TLV (expected to trigger ERR_UNSUPPORTED_TLV)

Header:
00 10 00 10 40 20 00 00 00 00

Unknown TLV Type=0x47
47 03 01 02 03

Payload: none

B.5.2 Malformed TLV length

Header:
00 10 00 10 40 20 00 00 00 00

TLV declares 10 bytes but only 4 follow:
22 0A 11 22 33 44

B.5.3 Oversized payload (>65535 bytes)

Omitted for brevity; implementers MUST reject any payload exceeding the 16-bit payload size constraint.

B.6 CoAP Blockwise Fragmentation

These examples illustrate partial ACP messages inside CoAP Block1 fragments.

B.6.1 Block 0

CoAP Header:
44 02 6A 77

Block1 Option:
2F 00 # NUM=0, M=1, SZX=0

Payload:
<first 64 bytes of encrypted ACP blob>

B.6.2 Block 1

CoAP Header:
44 02 6A 77

Block1 Option:
2F 11 # NUM=1, M=0, SZX=0

Payload:
<remaining bytes of encrypted ACP blob>

Receivers MUST reassemble before OSCORE decryption.

B.7 Summary

The test vectors in this appendix supplement those in Section 11 and are intended for parser verification, fuzzing, and interoperability testing. Implementations MUST be able to:

* Decode and validate all example headers.

* Correctly process TLVs, including multi-TLV messages.

* Reject malformed structures deterministically.

* Decrypt OSCORE ciphertexts after correct envelope processing.

* Reassemble Blockwise fragments prior to OSCORE processing.

These vectors serve as a baseline for ACP test suites and conformance validation frameworks.

Appendix C. Formal Semantics

This appendix provides a non-normative operational semantics for ACP, derived from the companion formal calculus that motivated the protocol. The semantics formalize the behavior of agents, message transitions, resource consumption, and subscription state. They are included for researchers, implementers building verified agents, and designers of static analyzers or runtime monitors.

C.1 Agent Configuration

A ACP agent is represented as a tuple:

A = (S, Q, R, C, B)

Where:
S : Local state of the agent (application-defined)
Q : Incoming message queue
R : Resource budget (CPU, memory, bandwidth)
C : Conversation table mapping correlation IDs to conversation state
B : Subscription table mapping topics to observers

Global system configurations consist of a multiset of agents and an abstract network buffer N:

A1 | A2 | ... | An ; N

Network delivery is modeled as nondeterministic but eventually fair unless congestion semantics dictate otherwise.

C.2 Message Syntax

Messages are modeled abstractly as:

```
M ::= PING(id, cid)
    | TELL(id, cid, tlvs, payload)
    | ASK(id, cid, tlvs, payload)
    | OBSERVE(id, cid, topic, filter)
```

The semantics below assume well-formed headers and TLVs; malformed messages trigger error semantics (Section C.7).

C.3 Operational Semantics for ACP Verbs

C.3.1 PING

PING models liveness detection and has no side effects on agent state except optional diagnostic logging.

```
A = (S, Q, R, C, B)
-------------------------------------------
A  A′ = (S, Q, R′, C, B)
```

Where R′ = R - cost(ping)

Figure 19: PING-RECV

The receiver MAY return a TELL error response, but this is not semantically required.

C.3.2 TELL

TELL conveys state and is idempotent under a given correlation ID. It updates the local knowledge state S according to application logic:

```
A = (S, Q, R, C, B)
-----------------------------------------
A  A′ = (S  payload, Q, R′, C, B)
```

Where:
R′ = R - cost(tell)
′′ denotes a monotonic, application-defined merge

Figure 20: TELL-UPDATE

C.3.3 ASK

ASK initiates a requestresponse conversation. Upon receiving an ASK, an agent enters a waiting state:

```
A = (S, Q, R, C, B)
cid  dom(C)
-----------------------------------------------------
A  A′ = (S, Q, R′, C[cid  wait(action)], B)

Where R′ = R - cost(ask)
```

Figure 21: ASK-START

```
C[cid] = wait(action)
S  action  value
-----------------------------------------------------
A  A′ = (S, Q ∪ {TELL(cid, value)}, R′, C′, B)

Where:
  C′ = C[cid  done]
  R′ = R - cost(eval)
```

Figure 22: ASK-RESPOND

ASK induces a deterministic two-step conversation unless cancelled or timed out.

C.3.4 OBSERVE

OBSERVE creates a persistent subscription represented in the agent configuration.

```
A = (S, Q, R, C, B)
topic  dom(B)
-----------------------------------------------------------
A  A′ = (S, Q, R′, C, B[topic  {cid}])

Where R′ = R - cost(observe)
```

Figure 23: OBS-REGISTER

Notifications are produced when internal state changes satisfy the trigger condition:

```
B(topic) = CIDs
S  S′ with event e matching filter
-----------------------------------------------------------
A  A′ with Q′ = Q ∪ { TELL(cid, e) | cid ∈ CIDs }
```

Figure 24: OBS-NOTIFY

C.4 Correlation Table Semantics

The correlation table C tracks conversation state with the invariant:

dom(C) MAX_CONV
cid uniqueness: cid dom(C) for new conversations

Expiration semantics remove entries automatically after a timeout:

C[cid] = wait(x)
expired(cid)

C′ = C \ {cid}

Figure 25: CID-EXPIRE

C.5 Resource Semantics

The resource model associates a cost with each transition.

cost : Transition →

A transition is only admissible if the agent has sufficient budget:

R cost(t)
 A ; N A′ ; N′

Transition allowed

Otherwise: A blocks or drops the message

Figure 26: RESOURCE-ADMISSIBILITY

Resource recharging (e.g., periodic replenishment) is allowed but not modeled directly in these semantics.

C.6 Fragmentation and Reassembly

Fragmentation is represented as a sequence of partial messages delivered independently:

frag(M) = [m0, m1, ..., mk]

Reassembly occurs only when all fragments are received:

∀i. mi ∈ N
---
reassemble(frag(M)) = M

Figure 27: REASSEMBLE

OSCORE decryption applies only after full reassembly; this preserves security invariants.

C.7 Error Semantics

Malformed messages or security violations produce error transitions.

C.7.1 Malformed TLV

invalid_tlv(M)

A A′ = (S, Q ∪ {TELL(cid, ERR_MALFORMED_TLV)}, R′, C, B)

Figure 28: ERR-MALFORMED

Unauthorized Action

not authorized(action, S)

A TELL(cid, ERR_UNAUTHORIZED)

Figure 29: ERR-UNAUTHORIZED

Version Downgrade Attempt

ver(M) < ver(A)

A TELL(cid, ERR_VERSION_MISMATCH)

Figure 30: ERR-DOWNGRADE

Summary

The semantics in this appendix capture the behavior of ACP agents, including the message lifecycle, conversation tracking, resource limits, fragmentation, error handling, and observable side effects. These rules closely reflect the informal operational description in the main sections of this specification and serve as a foundation for formal verification, conformance testing, and correct-by-construction implementations.

Acknowledgments

The design of ACP benefited from feedback across multiple research and engineering communities working on IoT systems, multi-agent communication, and distributed protocol design. The authors acknowledge the valuable insights provided by early reviewers, prototype implementers, and colleagues who explored ACP in constrained-device testbeds.

Special thanks are extended to members of the open-source contributors who reviewed early drafts of the ACP calculus and provided implementation reports via the project repository. Their feedback led to refinements in the state machines, TLV model, and transport bindings.

The authors also thank participants from constrained-network and OSCORE working groups whose discussions influenced the treatment of fragmentation, replay protection, and authentication in this specification.

This specification incorporates lessons from deployments in microcontroller-based sensing systems, autonomous control nodes, and large-scale telemetry environments. The authors acknowledge these deployments for motivating the resource model and deterministic behavior guarantees underlying ACP.

This work is an independent contribution and does not represent the views of any organization or government entity.

Authors' Addresses

Arnab Mallick
Centre for Development of Advanced Computing (CDAC)
Hyderabad
India
Email: arnabm@cdac.in
URI: https://arnab-m1.github.io

Indraveni Chebolu
Centre for Development of Advanced Computing (CDAC)
Hyderabad
India
Email: indravenik@cdac.in

Authors' Addresses

Arnab Mallick
Centre for Development of Advanced Computing (CDAC)
Hyderabad
India
Email: arnabm@cdac.in

Indraveni Chebolu
Centre for Development of Advanced Computing (CDAC)
Hyderabad
India
Email: indravenik@cdac.in