Journal of Information Systems Engineering and Management
2025, 10(59s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Toward a Reliable IoT Communication Protocol: UPPAAL-Verified Timed Automata for CoAP

Khalida Farida Khelili[1], laid Kahloul[2] , Abdelkarim Tahari[3]
[1] *Universite Amar Tlidji de Laghouat, Algeria* khelili.farida@univ-ouargla.dz,
[2] *Universite Med khieder . Biskra , Algeria* l.kahloul@univ-biskra.dz
[3] *Universite Amar Tlidji de Laghouat, Algeria* k.tahari@lagh-univ.dz

ARTICLE INFO

Received: 30 Dec 2024

Revised: 19 Feb 2025

Accepted: 27 Feb 2025

ABSTRACT

In our world, the Internet of Things (IoT) has long been deeply embedded in daily life, linking devices across domains as varied as healthcare, transportation, and smart homes. This interconnected world relies on lightweight communication protocols to operate under tight energy and resource constraints. Among them, the Constrained Application Protocol (CoAP) stands out for its simplicity and HTTP-like semantics—yet questions remain about its reliability and efficiency at scale and on lossy networks.

In this work, we build a formal model of CoAP using timed automata in UPPAAL [2]. The model captures key aspects of the protocol—confirmable and non-confirmable messages, acknowledgments, retransmissions, timeout management, and token/MID correlation—and is intended first for validation of functional correctness (safety, liveness, reachability). To expose timing effects without altering endpoint logic, we compose the client and server with a minimal network template. Beyond the core specification [1], we design extension hooks to broaden validation and later evaluation: Observe for notification ordering and cancel safety [3], Block-Wise transfers for block progression and reassembly completeness [4], and Hop-Limit for loop-freedom via strict hop decrement and reset on exhaustion [5]. Our study not only delivers a validated formal model of CoAP but also illuminates its strengths, limitations, and opportunities for refinement, paving the way toward more scalable and reliable IoT communication.

Keywords: IoT, IoT communication protocols, CoAP, UPPAAL, timed automata, formal verification, validation.

1 INTRODUCTION

Everyday objects now converse quietly in the background—thermostats learn our habits, buses announce their positions, soil sensors nudge irrigation systems. This pervasive Internet of Things (IoT) places tiny, battery-powered devices in environments that are noisy, lossy, and resource-constrained. In such conditions, communication must be not only lightweight but trustworthy: a missed acknowledgment wastes scarce energy; a rare timing glitch can stall an actuator at the wrong moment. The Constrained Application Protocol (CoAP) adapts the familiar REST/HTTP style to constrained networks and is widely used in practice [1], within a broader wave of IoT standardization [6]. Deployments commonly rely on well-known extensions—Observe for event notifications, Block-Wise transfers for large objects, and Hop-Limit to avoid forwarding loops—which shape behavior in the field [3–5].

Even with this momentum, ensuring reliable behavior at scale remains difficult. Wireless links burst, buffers back up, batteries fade, and subtle timing interactions matter. Empirical studies quantify latency, loss, throughput, and energy under varied workloads and stacks, often comparing CoAP with alternatives [7–13]. Formal analyses aim at a different guarantee: machine-checked properties such as safety, liveness, and conformance, using model checkers and process algebras including SPIN, CSP/PAT, and UPPAAL [14–16]. What remains uncommon is a reusable

Journal of Information Systems Engineering and Management
2025, 10(59s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

artifact that is faithful to the core specification, focuses squarely on endpoint timing rules, and is simple enough to extend without rewriting the model.

This paper contributes such an artifact: an executable formal model of base CoAP in classic UPPAAL using timed automata, derived directly from the normative specification [1]. To keep the model compact and to isolate endpoint correctness, we do not introduce a separate network automaton. Instead, all variability relevant to reliability is captured at the endpoints: the server issues acknowledgments after **a** bounded processing delay, and the client enforces specification-consistent timeouts with exponential backoff. In this abstraction, duplicates arise naturally when early backoff windows expire and the client retransmits; the server's deduplication and token/MID discipline resolve them, with NSTART limiting concurrent confirmable exchanges.

On this basis, we verify properties practitioners care about—ACK/CON matching, token–MID binding, bounded retransmissions, and deadlock freedom—and we establish timing-liveness under a transparent bounded-delay assumption (D_ACK_MAX < RTO(MAX_RETRANSMIT)) in classic (non-probabilistic) UPPAAL. When assumptions are violated or rules are weakened, the model produces concrete counterexample traces that expose where and how failures occur (e.g., missing deduplication or premature token reuse), supporting both diagnosis and pedagogy, we have add a compact network model with delay bounds (and optional duplicates) which can let us compute a precise progress condition that users can check quickly.

Although our focus is the formal validation of the base protocol, the model is designed to grow. We outline clean extension hooks aligned with widely used enhancements—Observe, Block-Wise, and Hop-Limit [3–5]—so the same core can later support analyses of notification ordering, transfer completeness, and loop-freedom without altering foundational components. The result is a rigorous, readable, and reusable starting point for validating CoAP's timing behavior today, and for extending that validation in future work.

The remainder of this paper is organized as follows. Section 2 reviews the state of the art on IoT protocols and CoAP, section 3 presents the related work, section 4 presents UPPAAL and the formal method of modeling CoAP with UPPAAL framework. Section 5 presents the results and the creation of CoAP model, and section 6 presents the formal verification of the protocol, at the end we conclude the paper and we give the directions for future research.

2 STATE OF ART:

2.1 IoT Communication Protocols:

The IoT ecosystem relies on a variety of communication protocols designed to support devices operating under strict constraints of bandwidth, processing power, memory, and energy [5]. At the application layer, several lightweight solutions are widely adopted. MQTT (Message Queuing Telemetry Transport) employs a publish/subscribe paradigm optimized for low-bandwidth and high-latency networks, making it suitable for industrial IoT and large-scale sensor deployments. AMQP (Advanced Message Queuing Protocol) provides advanced queuing and routing services with strong reliability but incurs higher overhead, which limits its applicability in highly constrained environments. DDS (Data Distribution Service) supports real-time and mission-critical systems, offering strong reliability and scalability at the cost of greater resource requirements.

Within this landscape, the Constrained Application Protocol (CoAP) has become particularly relevant for constrained devices. Standardized by the IETF in RFC 7252 [2], CoAP extends the RESTful model of HTTP to IoT systems by operating over UDP with minimal message overhead, asynchronous exchanges, and support for multicast. Its design enables efficient resource discovery, simple proxy and caching functions, and integration with standard web services, making it a key protocol for M2M communication.

2.2 CoAP in IoT

The Constrained Application Protocol (CoAP) is a specialized web transfer protocol developed by the IETF to extend RESTful communication into constrained environments [21]. CoAP follows a client–server model, similar to HTTP, but introduces several technical adaptations to address the needs of IoT systems, figure 1 presents the architecture of CoAP.

Journal of Information Systems Engineering and Management
2025, 10(59s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

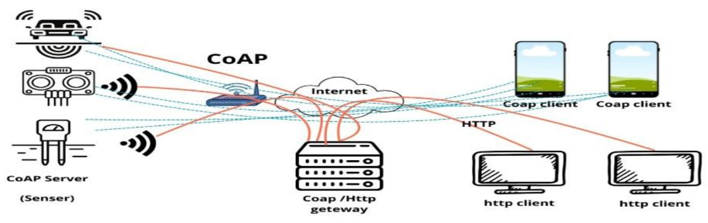


Figure 1: CoAP protocol

CoAP is designed to combine lightweight implementation with web interoperability, making it suitable for environments where energy efficiency, low latency, and robustness are required. It supports asynchronous communication over UDP with optional reliability, compact headers, URIs and content negotiation, proxying and caching, and a stateless mapping to HTTP [1]. For security, CoAP relies on Datagram Transport Layer Security (DTLS), and in extended scenarios, on Object Security for Constrained RESTful Environments (OSCORE) [21].

The CoAP protocol have many characteristics related to our work [1]:

✓ **Transport and Reliability:**CoAP is built on UDP, which provides low overhead and fast transmission suitable for resource-limited devices. To compensate for the lack of guaranteed delivery in UDP, CoAP defines four message types:

- Confirmable (CON): requires acknowledgment (ACK) or reset (RST), ensuring reliability.
- Non-confirmable (NON): sent without requiring acknowledgment, minimizing delay and energy.
- Acknowledgment (ACK): confirms receipt of a CON message.
- Reset (RST): indicates that a CON message was received but could not be processed.

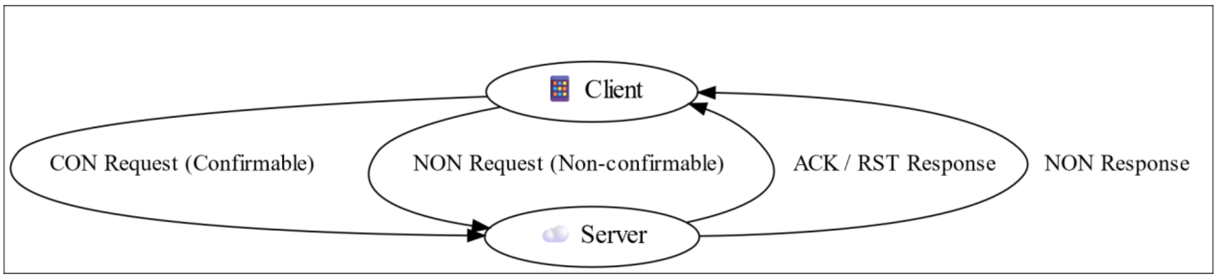


Figure 2: CoAP messages

This mechanism provides optional reliability, balancing between efficiency and robustness, and it depends on application requirements [2].

✓ **Message Format:** The CoAP message header is compact (4 bytes), it consists of a version field, message type, token length, code, and message ID. The token field is used to match requests and responses; it enables asynchronous transactions. There are Options (such as URI path, content type, or caching directives) follow a delta-encoded format that reduces header size and parsing complexity. This lightweight design makes CoAP well-suited for devices with limited memory and CPU.

✓ **RESTful Operations:** CoAP supports the four core REST methods: GET, POST, PUT, DELETE, enabling direct mapping to HTTP. Resources are identified by Uniform Resource Identifiers (URIs), and content negotiation is supported through Internet media types (e.g., JSON, CBOR, XML). This ensures compatibility with web technologies while remaining lightweight.

We present some key CoAP elements:

• Identifiers. Message ID (MID) for deduplication at the message layer; Token for request/response correlation. The Client stores Token on send; the Server echoes it; a property forbids token reuse while a request is pending.

• Reliability and timers. For CON, the Client retransmits on timeout with exponential backoff:

RTO(k) = ACK_TIMEOUT · 2^k · RAND_FACTOR_fixed.

The Client clock x measures backoff; the Server clock y models processing delay before ACK with: y ∈ [D_ACK_MIN, D_ACK_MAX].

• Duplicate handling. Duplicates arise when early timeouts trigger retransmissions. The Server maintains a small dedup cache keyed by (peer, MID); duplicates cause re-ACK without re-processing.

• Flow control. NSTART = 1 limits outstanding CON exchanges per pair.

• Error path. RST is modelled as an abort transition (no detailed taxonomy).

2.3 UPPAAL and its query language:

UPPAAL is a model checker for networks of timed automata—finite-state machines equipped with real-valued clocks. It is well suited to CoAP because retransmission timeouts, bounded server delays, and backoff policies are naturally expressed with clocks and guards [10].

✓ **Modeling essentials:** Endpoints are templates with locations and edges; clocks carry timing (client x, server y); invariants bound time in a location (e.g., y ≤ D_ACK_MAX); guards enable transitions (e.g., x ≥ RTO(k)); updates reset clocks and bounded integers; binary channels (a!/a?) synchronize handshakes (req_CON, ack_empty). Small bounded domains are used for counters, Message IDs, Tokens, and caches to keep the state space finite.

✓ **Query language:** UPPAAL supports a practical fragment of TCTL:

- A[] φ (φ holds on all paths, globally: safety/invariant)
- E<> φ (there exists a path where φ eventually holds: reachability)
- A<> φ (on all paths, φ eventually holds: liveness under assumptions)
- deadlock / not deadlock (built-in predicates)

3 RELATED WORKS

The Constrained Application Protocol (CoAP) is a lightweight, REST-style protocol standardized for constrained environments and widely used in practice [1]. Deployments often rely on well-known extensions—Observe for event notifications, Block-Wise transfers for large payloads, and Hop-Limit to avoid forwarding loops—which shape behavior in the field [3–5]. For modeling real-time behavior, we follow the timed-automata tradition with UPPAAL as a mature verification environment [2].

A first thread of related work develops formal models of CoAP to obtain guarantees that testing alone cannot provide. Using PROMELA/SPIN, Agarwal et al. verify safety and liveness for CoAP layered over RPL; this is informative for routing interactions but remains untimed, so retransmission clocks and ACK timing are not first-class elements [7]. In a complementary direction, CSP/PAT has been used to verify group and enhanced-group communication patterns for CoAP, clarifying ordering and progress in group scenarios rather than the base request/response core [8, 9]. Within the UPPAAL family, UPPAAL-SMC has been applied to time-bounded properties with probabilistic semantics [10], whereas our focus is on classic (non-probabilistic) UPPAAL to prove endpoint invariants, deadlock-freedom, and timing-liveness for the base protocol. Methodologically, work on UPPAAL property-specification patterns and on IoT/CPS case studies (e.g., translating process calculi to timed automata for smart-home scenarios) demonstrates both the expressiveness of the formalism and good practices for writing checkable requirements [11, 12]. At the boundary with implementations, MPInspector mines finite-state machines from protocol stacks—including CoAP—

to reveal deviations and vulnerabilities; this is complementary to our approach, which supplies an executable specification of the intended timed behavior [13].

A second thread provides empirical perspective on performance and deployment. Comparative studies and testbeds report that CoAP is lightweight and fast in one-hop or device–gateway settings, but can be sensitive to bursty traffic and loss; results also position CoAP favorably against HTTP and competitive with MQTT/MQTT-SN in latency and energy [14–18]. With security enabled, evaluations show that DTLS/OSCORE introduce overhead yet remain practical for constrained devices, and comparisons with HTTPS/DTLS/OSCORE mixes help position CoAP in secure device communication [19, 20]. We do not reproduce these measurements; rather, we use them to motivate a mechanized validation of time-critical rules at the endpoints that we model.

Positioning. The gap addressed here is a reusable, executable, classic-UPPAAL model of the base CoAP core—faithful to the standard and focused on endpoint timing semantics (CON/NON, ACK/RST, exponential backoff, Token/MID discipline, deduplication, NSTART). The contribution is a proof-oriented artifact that establishes safety, deadlock-freedom, and timing-liveness under explicit bounded-delay assumptions. Observe, Block-Wise, and Hop-Limit [3–5] are treated as future overlays, keeping the present scope precise and aligned with what is actually verified.

More generally, recent research demonstrates the growing use of **UPPAAL** in IoT and cyber-physical systems (CPS). Chen and Zhu [22] modeled smart home scenarios by extending the CaIT calculus into UPPAAL timed automata, successfully verifying temporal synchronization properties. Vogel et al. [23] developed a catalog of property specification patterns for UPPAAL. They are simplifying the translation of real-time requirements into verifiable automata.

Comparative Summary of Related Work on CoAP:

Formal work addresses that generality requirement by supplying machine-checked guarantees over all admissible executions. Models in SPIN establish safety and liveness for CoAP layered over routing, but timing remains implicit and retransmission clocks are not first-class entities [7]. CSP/PAT verifies group and enhanced-group semantics, clarifying ordering and progress in those scenarios rather than in the base request–response core [8, 9]. UPPAAL-SMC introduces probabilistic timing and supports statistical estimation of time-bounded properties, yet typically targets quantitative assessment rather than proof obligations in the classic sense [10]. Methodological contributions and case studies further motivate timed automata and UPPAAL for real-time IoT logic [11, 12], while implementation-mining approaches such as MPInspector expose divergences and vulnerabilities in practice [13].

Positioned at this intersection, the present work contributes an executable model of the base CoAP specification in classic UPPAAL, making endpoint timing semantics explicit—CON/NON exchanges, ACK/RST handling, exponential backoff, token/MID discipline, deduplication, and NSTART—and proving safety, deadlock-freedom, and timing-liveness under transparent bounded-delay assumptions. The model is derived from the normative standard [1] and is structured for subsequent overlays aligned with Observe, Block-Wise, and Hop-Limit [3–5], while drawing on established UPPAAL practice for property authoring and verification [2, 11, 12].

Table 1: Comparative Summary of Related Work on CoAP:

Ref / Authors	Method/Tool	Focus	Limitations	Gap / Remarks
[7] Agarwal et al. (2016)	SPIN / PROMELA	CoAP over RPL; safety & liveness	Untimed; no retransmission clocks or ACK timing	Formal validation only; does not capture endpoint timing rules
[8] Chen, Li & Zhu (2022)	CSP / PAT	Group CoAP correctness	Group semantics; no retransmission modeling	Complements base CoAP core; different surface than this paper
[9] Chen, Zhu & Yuan (2023)	CSP / PAT	Enhanced-group CoAP	Same group focus; no delay/energy properties	Orthogonal to our scope; possible future overlay

[11] Chen & Zhu (2023)	UPPAAL timed automata (IoT smart home)	Temporal synchronization in IoT/CPS	Domain-specific; not protocol-focused	Supports choice of UPPAAL/timed automata for real-time logic
[12] Vogel, Carwehl, Rodrigues & Grunske (2022)	UPPAAL property patterns	Catalog for real-time properties	Methodology only; no CoAP model	Guides property authoring (safety, liveness, deadlock)
[13] Wang et al. (2021)	MPInspector	FSM extraction; vulnerability analysis	Coverage-dependent; not formal proofs	Complementary; our model is an executable spec baseline
[14] Gündoğan et al. (2018)	Testbed	NDN/CoAP/MQTT comparison	Measurement-only; not exhaustive	Shows burst sensitivity; motivates formal timing validation
[16] Bansal & Priya (2020)	Simulation	MQTT vs CoAP across simulators	Simulation-only; no real devices	Comparative figures; no correctness validation
[17] Silva, Carvalho, Soares & Sofia (2021)	FIT-IoT testbed	MQTT/CoAP/OPC UA benchmarking	Environment-specific	Supports CoAP's lightweight design; no formal analysis
[18] Seoane, Alberti, Militano & Iera (2021)	Testbed	CoAP & MQTT with security enabled	Performance focus only	Lacks timed correctness; complementary to our validation
[19] Andersen & Dalsgaard (2023)	Secure device comms eval	CoAP, OSCORE, DTLS, HTTPS	Security-centric; device-level	Confirms viability; no formal timing proofs
[20] Palombini, Tiloca & Palombini (2021)	OSCORE performance	IoT measurements of OSCORE	Measurement-based	Practical results; lacks formal endpoint guarantees

There is no prior study has unified formal correctness validation with performance evaluation in a single framework. Our work addresses this gap by building a UPPAAL timed automata model of CoAP that integrates functional correctness with key performance metrics, enabling rigorous analysis and generating perspectives for future protocol enhancement.

Despite the body of work, performance evaluation and formal verification remain largely separate research tracks. Simulations and testbeds address throughput, delay, and energy efficiency but lack exhaustive guarantees, while formal models verify correctness or security without integrating performance aspects. To date, no study has combined formal validation with quantitative performance metrics—such as packet loss, delay, throughput, and energy consumption—in a unified framework. Our work addresses this gap by developing a timed automata model of CoAP in UPPAAL that enables both functional verification and performance evaluation, ultimately identifying perspectives for protocol enhancement.

4 MODULIZATION METHOD

To validate a protocol, it must first be modeled using a formalism chosen according to three key criteria: (i) its ability to capture the essential characteristics of the system under study, (ii) its modeling power, i.e., how simply and clearly these characteristics can be expressed, and (iii) its analysis power, meaning the capacity to formally express and verify properties of the system. Once the model is defined, model checking enables the exhaustive exploration of all possible execution paths. The system constraints are first specified as formal properties, covering both behavioral aspects (e.g., message reliability, retransmissions) and temporal aspects (e.g., delays, timeouts). A set of representative scenarios is then constructed, and the model-checking engine verifies whether these properties hold across all executions.

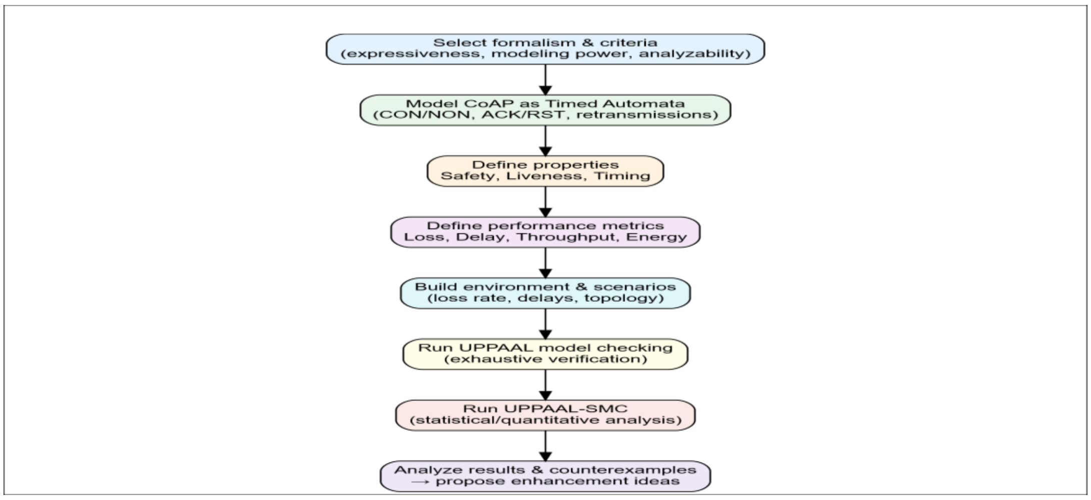

Select formalism & criteria (expressiveness, modeling power, analyzability)
Model CoAP as Timed Automata (CON/NON, ACK/RST, retransmissions)
Define properties Safety, Liveness, Timing
Define performance metrics Loss, Delay, Throughput, Energy
Build environment & scenarios (loss rate, delays, topology)
Run UPPAAL model checking (exhaustive verification)
Run UPPAAL-SMC (statistical/quantitative analysis)
Analyze results & counterexamples → propose enhancement ideas

Figure 3 : CoAP Formel modeling steps (UPPAAL)

In our case, the Constrained Application Protocol (CoAP) is modeled as a set of timed automata within the UPPAAL framework. This formalization captures both the functional behavior of CoAP (confirmable and non-confirmable message exchanges, acknowledgments, retransmissions) and its temporal dynamics (timeouts).The specified properties include safety, liveness, and timing constraints, as well as performance-related measures such as packet loss, delay, throughput, and energy consumption can be measured too in a future work. Through the UPPAAL model-checking engine, we are thus able to carry out a rigorous formal validation of CoAP, bridging correctness and performance analysis in a unified approach.

in Figure 3 we present the Step-by-step workflow of CoAP modeling and verification in UPPAAL. The process begins with selecting an appropriate formalism and modeling CoAP as timed automata. Properties (safety, liveness, timing) are then specified. Representative scenarios are constructed, followed by exhaustive verification with UPPAAL and quantitative analysis with UPPAAL-SMC can be done. The results guide the identification of limitations and perspectives for protocol enhancement.

4.1 UPPAAL modulization:

An **UPPAAL model** is based on *timed automata*, which combine finite-state machines with real-time clocks. The description of such a model is generally divided into three main parts:

1. **Global and local declarations** – These include variables, clocks, and constants that describe the state of the system and timing constraints. In the case of CoAP, global declarations define retransmission counters, timeout values, and message identifiers, while local declarations are used for automata-specific variables such as timers for client or server processes.

2. **Automata templates** – Each component of the protocol is represented as an automaton template, describing its possible states and transitions. For CoAP, templates model the **client**, **server**, and **communication channel**, capturing behaviors such as sending confirmable (CON) or non-confirmable (NON) messages, receiving acknowledgments (ACK), and handling retransmissions or resets (RST).

3. **System definition** – This part specifies how the different templates are instantiated and composed into a complete system. In our CoAP model, the system definition links the client automaton, server automaton, and channel, enabling interaction between them and allowing the model checker to explore all possible execution paths.

In our work have realize this configuration:

a. Declarations

The declaration section in UPPAAL defines the variables, clocks, constants, and communication channels that will be used by the automata. These declarations can be either global, shared across all templates, or local, specific to a single automaton. They form the foundation of the model, allowing the system to represent both protocol behavior and timing constraints [25].

- **Global declarations** are used to define system-wide elements. In the CoAP model, these include:
  - **Clocks**, which represent retransmission timers, response deadlines, and delay counters.
  - **Constants**, such as the maximum number of retransmissions or fixed timeout intervals.
  - **Channels**, which synchronize communication between the client, server, and channel automata (e.g., for requests, acknowledgments, or resets).
  - **Global variables**, including message identifiers, counters, and flags to represent packet delivery or loss.
- **Local declarations** are specific to each automaton template. For instance :
  - The **client automaton** declares variables to track tokens, retransmission attempts, and the current request state.
  - The **server automaton** manages variables related to received messages and responses.
  - The **channel automaton** defines variables for packet loss probability and transmission delays.

b. Automata Templates

In UPPAAL, the behavior of each system component is represented through automata templates, which define the possible states of the component and the transitions between them. Each template may also include local variables and clocks to refine its operation. This modular design makes it possible to represent complex communication systems in a structured and analyzable way [10].

For the CoAP protocol, we developed two main automata templates:

- **Client Automaton** – Models the behavior of a CoAP client initiating communication. States include *Idle*, *Sending Request*, *Waiting for Response*, *Retransmitting*, and *Completed*. Transitions occur when the client sends either a confirmable (CON) or non-confirmable (NON) request through the synchronization channel req. If the expected acknowledgment is not received within the timeout interval, a retransmission is triggered, increasing the retransmission counter until either an acknowledgment is received or the maximum retry threshold is reached.
- **Server Automaton** – Represents the CoAP server handling incoming requests. States include *Idle*, *Processing Request*, and *Sending Response*. Upon synchronization with a req event, the server processes the request and replies with either an acknowledgment (ack) for confirmable messages, a NON response for non-confirmable requests, or a reset (rst) in case of errors.
- To separate endpoint logic from transport variability, we compose the client and server with a minimal link model that adds bounded one-way delay (and, optionally, loss/duplicates) without changing endpoint code.

Our modeling ensures that both the functional aspects of CoAP (client–server communication) and the environmental conditions are accurately represented. Such modularity and timing precision are among the strengths of UPPAAL in protocol validation [10].

c. System Definition

Once the automata templates are constructed, UPPAAL requires a system definition that specifies how the different templates are instantiated and executed together. This step integrates the separate components into a single model where their interactions can be formally analyzed [21].

In our CoAP model, the system consists of three interacting automata: the Client, the Server, and the Channel.

- The **Client** automaton generates CoAP requests and manages retransmissions.
- The **Server** automaton processes these requests and generates the corresponding responses.

- The **Channel** automaton introduces network dynamics such as message delays and losses.

The system definition in UPPAAL instantiates these templates and executes them in parallel composition, with message exchanges coordinated via synchronization channels (req, ack, rst). An excerpt of the definition is given below in figure 4:

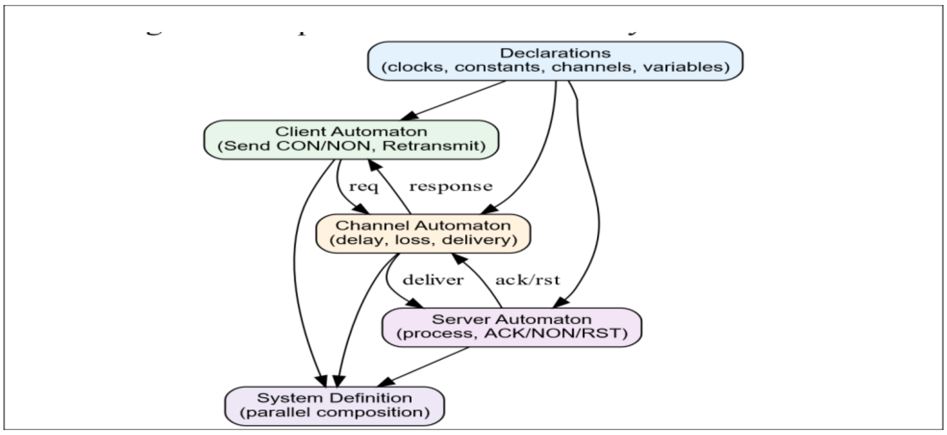


Figure 4: simplified UPPAAL CoAP system structure

This composition instructs UPPAAL to execute the automata concurrently, ensuring that communication and timing constraints are respected across all components. By combining client behavior, and server responses, the system definition creates a realistic yet analyzable framework for validating CoAP in IoT environments [10].

UPPAAL query	**Purpose**
A[] !Observer.badAck	No ACK occurs without a prior matching CON (safety).
A[] !Observer.tokenReuseWhilePending	A token is never reused while its request is still pending (safety).
A[] Observer.duplicates_per_mid <= 1	At most one side effect per Message ID; duplicates are re-ACKed only (safety).
A[] not deadlock	The system never reaches a state where time cannot progress and no transition is enabled.
` A<> (Client.Done || Client.Abort)	progress under bounded delay

Table 2: UPPAAL queries

For the server's maximum reply delay must be strictly smaller than the final retransmission:

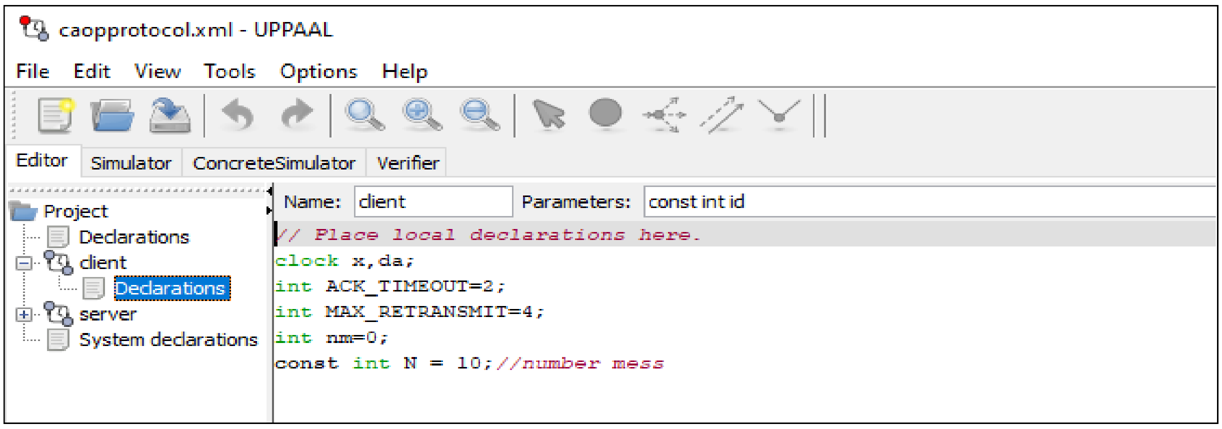
D_ACK_MAX < ACK_TIMEOUT * 2^MAX_RETRANSMIT * RAND_FACTOR_fixed.

5 RESULTS

5.1 Creation of CoAP model:

In our system we have two parts : CoAP template Client and CoAP template server.

In figure 5 , we presents the initial declarations for the test and validations:



Name: client Parameters: const int id

```
// Place local declarations here.
clock x,da;
int ACK_TIMEOUT=2;
int MAX_RETRANSMIT=4;
int nm=0;
const int N = 10;//number mess
```

Figure 5: CoAP declaration

- **CoAP client template:**

This template resumes the work of CoAP client. Inspire from RFC 7252 [1]. This figure represents a timed automata model of CoAP communication in UPPAAL. We have formalized how different types of CoAP messages (CON, NON; GET, POST, PUT, DELETE) are processed, the retransmissions are handled through waiting states, and the message exchanges terminate. Our model helps us to relies the next step of our work, which is the formal validation of CoAP performances.

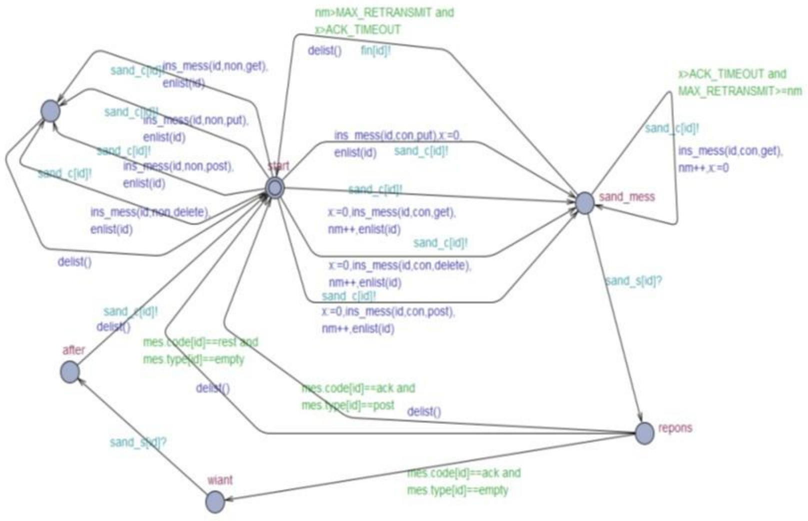


Figure 6: CoAP client Template

In our Model Captures , the Message Types witch distinguishes between CON and NON messages, each with different handling rules (acknowledgment vs. no acknowledgment), the Message Codes which Supports CoAP operations (GET, POST, PUT, DELETE), the Timeout & Retransmission , they are presented by the confirmable messages into the wait state, it is modeling delays and retransmissions, and the End-to-End Exchange which presents the lifecycle of a message transaction(from star to after).

structure of the Model

- States such as start, receive_mess, receive_con, receive_non, wait, fin_mess, and after represent protocol stages.
- Transitions are annotated with guards (conditions in green), synchronizations (blue, e.g., sand_c[front]!), and updates (variable assignments).

Key States:

start → entry point of the automaton. The initial communication begin by sending a CoAP message (sand_c[e]?).

1. **receive_mess** → the central state handling different incoming CoAP requests ((mes.code[front()]), distinguishing GET, POST, PUT, DELETE, and whether it is CON or NON.)

2. **receive_con** vs **receive_non** : **receive_con:** handles confirmable messages: they require ACKs, and retransmissions are possible, and the receive_non handles non-confirmable messages: they do not require acknowledgments.

3. **Wait** : Represents a timeout period where the automaton waits for an acknowledgment or response. If ACK is not received in time, retransmission is triggered, the limite of retransmission is 4.

4. **fin_mess** : successful processing of a message. It ensures the client/server transitions into a stable state after processing.

5. **After** : terminal state, which represents the end of an exchange or protocol termination.

The summar of important transition in table 3:

From → To	Guard	Sync / Update (excerpt)	Semantics
start → sand_mess (CON, any method)	—	ins_mess(con, method); enlist(id); sand_c[id]!; x:=0; nm:=0	Create CON and open first backoff window.
start → start (NON, any method)	—	ins_mess(non, method); enlist(id); sand_c[id]!; delist()	Send NON; no waiting.
sand_mess → sand_mess	x ≥ ACK_TIMEOUT ∧ nm < MAX_RETRANSMIT (or x ≥ RTO[nm])	ins_mess(con, method); nm:=nm+1; x:=0; sand_c[id]!	Timeout and retransmit.
sand_mess → start	x ≥ ACK_TIMEOUT ∧ nm = MAX_RETRANSMIT (or x ≥ RTO[nm])	delist()	Abort after final window.
sand_mess → repons	sand_s[id]?	—	A reply arrived; classify it.
after → start	—	—	Ready for the next request.

Table3 : Example of CoAP client transition

B. CoAP server template:

After declaration we have create this model:

Incoming messages are taken from the head of the queue (front()), then dispatched by class and method. **NON** requests elicit immediate **NON** responses and proceed to fin_mess. **CON** requests either return a **piggybacked** response (ACK+code → fin_mess) or take the **separate-response** path: send an Empty ACK, enter wait, later send a CON response, and finish when the client ACKs it. The unknown inputs yield rest (RST). Notation: x? = receive, x! = send; ins_mess(...) enqueues the outbound message; the sand_s[...]! transmits; fin_mess/after perform cleanup

Journal of Information Systems Engineering and Management
2025, 10(59s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

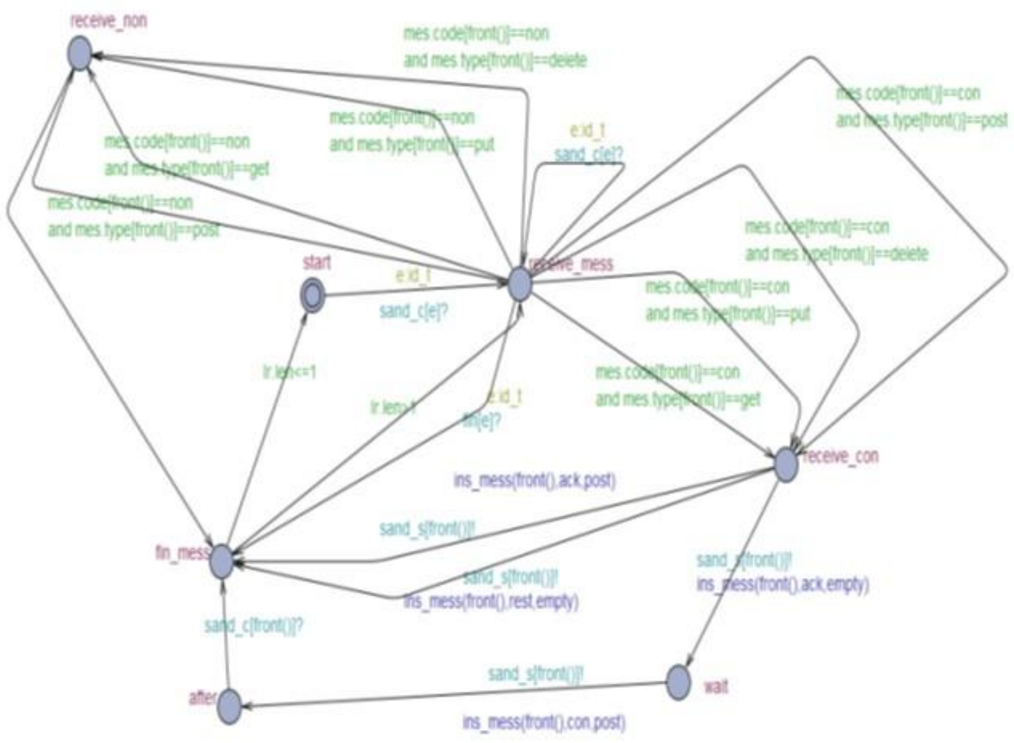

receive_non
mes.code[front()]==non and mes.type[front()]==delete
mes.code[front()]==con and mes.type[front()]==post
e:id_t sand_c[e]?
mes.code[front()]==non and mes.type[front()]==put
mes.code[front()]==non and mes.type[front()]==get
mes.code[front()]==non and mes.type[front()]==post
start
e:id_t sand_c[e]?
receive_mess
mes.code[front()]==con and mes.type[front()]==delete
mes.code[front()]==con and mes.type[front()]==put
lr.len<=1
lr.len>1
e:id_t fin[e]?
mes.code[front()]==con and mes.type[front()]==get
receive_con
ins_mess(front(),ack,post)
sand_s[front()]!
fin_mess
sand_s[front()]! ins_mess(front(),rest,empty)
sand_s[front()]! ins_mess(front(),ack,empty)
sand_c[front()]?
after
sand_s[front()]!
wait
ins_mess(front(),con,post)

Figure 7: CoAP server Template

The automaton processes one inbound message at a time from the head of the input queue (front()), under the guard lr.len>0. At start, the server consumes the event sand_c[e]? (client → server), binds e:id_t, and moves to receive_mess, where the message is classified by class and method using mes.code[front()] ∈ {non,con,ack,rest} and mes.type[front()] ∈ {get,put,post,delete,empty}.

The summary of the transition in the model :

From → To	**Guard on head (front())**	**Sync / Update**	**Semantics**
start → receive_mess	—	sand_c[e]?	A request arrives; enqueue and classify.
receive_mess → receive_con	mes.code==con ∧ mes.type∈{get,put,post,delete}	—	Confirmable path.
receive_mess → receive_non	mes.code==non ∧ mes.type∈{get,put,post,delete}	—	Non-confirmable path.
receive_con → fin_mess	—	sand_s[front()]!; ins_mess(front(), ack, post)	Piggyback ACK with response.
receive_con → wait	—	sand_s[front()]!; ins_mess(front(), ack, empty)	Empty ACK; prepare for separate response.
after → start	—	—	Ready for the next burst.

Table 4: Transition of server model

At the end the two models are attached to make the verification.

In conclusion, the client and server templates give a clear, executable picture of how CoAP behaves at the endpoints. The client sends NON or CON requests, waits sensibly, and retries when timeouts hit; the server receives them, answers either immediately (piggyback) or in two steps (empty ACK then a later response), and keeps a simple queue so nothing is lost. They talk over the same indexed channels, so each exchange stays neatly paired from send to reply. This small composition is easy to read and run, yet strong enough to check what matters in practice: ACKs match requests, retries are bounded, the system never deadlocks, and—under a simple timing bound—every exchange eventually finishes. Just as important, the structure is ready to grow: we can drop in a bounded server-delay, add deduplication, or layer Observe/Block-Wise/Hop-Limit without rewriting the core.

C. Network stub (delay / optional duplicate):

We interpose a lightweight link automaton in each direction (client→server; server→client). Each link stores the message identifier, waits for a nondeterministic delay $d \in [Dmin,Dmax]$, delivers once, and—if enabled—may emit a single duplicate after an additional $d' \in [DUPmin,DUPmax]$. This keeps endpoints unchanged while exposing timing that matters for reliability.

Progress condition (updated):

$$Dmax\text{^}CS + Dmax\text{^}svc + Dmax\text{^}SC < RTOmax\text{^}C < RTOmax.$$

We compose the client and server with a tiny network stub made of two identical one-way links (request and reply). Each link adds only what we need for timing: a bounded delivery delay [Dmin,Dmax] for the first copy and, if enabled, the possibility of exactly one duplicate within a short window [DUPmin,DUPmax]. An optional "loss" switch lets the link drop a packet; for liveness claims we keep loss off (standard fairness). The endpoints themselves are unchanged—we simply route their existing send/receive channels through these links—so all safety properties (ACK/CON matching, bounded retries, deadlock freedom) are preserved. With this composition, progress becomes an explicit timing rule: the sum of worst-case delays on the client→server link, server processing, and server→client link must be strictly less than the client's final retransmission window to guarantee completion.

Property / Query	**Endpoint-only**	**Networked**
ACK/CON matching (A[] !badAck)	✓	✓
Bounded retries (A[] k ≤ MAX_RETRANSMIT)	✓	✓
NSTART discipline	✓	✓
Deadlock freedom (A[] not deadlock)	✓	✓
Liveness under bound	✓ (with Dmaxsvc<RTOmax	✓ (with DmaxCS+Dmaxsvc+DmaxSC<RTOmax

Table5: Network property

At equality, a last ACK can coincide with the final timeout, and the model may abort—explaining precisely where liveness fails.

D. Extending the Core Model: Block-Wise, Observe, Hop-Limit:

We outline three thin overlays—Block-Wise transfers [4], Observe [3], and Hop-Limit [5]—that reuse the existing channels and control flow, adding only a few integers/buffers and a focused set of properties without disturbing the core proofs :

Block-Wise transfers [4] : add per-exchange block state (num,m,szx) and a bounded reassembly buffer keyed by Token; we then verify monotone block numbering, “one effect per object,” bounded buffer usage, and completion when each block is acknowledged.

Observe [3]: introduces a server subscription table keyed by (client,Token) (\text{client},\text{Token}) (client,Token) and a notifier that emits updates with non-decreasing Observe sequence numbers; we verify “at most one active subscription per token,” order preservation at the client, and clean cancel (no notifications after cancel).

Hop-Limit [5]: adds an integer hop counter decremented on each forward (or emulated proxy hop); we verify “no forward at zero,” loop-freedom, and bounded forwarding depth. These overlays touch only a few integers and a small buffer, preserve our core safety proofs, and open a path to quantitative evaluation without restructuring the model.

Extension (cite)	**Minimal model additions**	**Properties to check**
Block-Wise [4]	blk_num, blk_m, blk_szx; bounded reassembly buffer (by Token)	Monotone blocks; one effect per object; buffer bound; completion if each block ACKed
Observe [3]	Server subscription table (client,Token); notifier with Observe seq	One active subscription per token; non-decreasing sequences; no notifications after cancel
Hop-Limit [5]	hop_limit field; decrement on each forward (emulated proxy chain)	No forward at zero; loop-freedom; bounded forwarding

Table6: Extending the Core Model: Block-Wise, Observe, Hop-Limit

In table the most important proprieties to check , when madding those changes in the CoAP protocol and modeling the extending core model- according to the propositions of new rfc [3-5].

6 FORMAL VERIFICATION

6.1 Verification plan:

in our formal verification we check what is important for a CoAP endpoint pair (client + server):

- Safety (always true):

1. ACK/CON matching — no ACK without a prior matching CON.
2. Bounded retries — the client never exceeds MAX_RETRANSMIT.
3. Flow control — NSTART is respected (at most one CON in flight).
4. Deadlock freedom — the model never gets stuck.

- Liveness (eventual completion under a timing bound):

Every exchange eventually finishes (reply or clean abort) assuming the server’s maximum reply time is strictly below the client’s final retransmission:
D_ACK_MAX < RTO_MAX, where RTO_MAX = ACK_TIMEOUT × 2^MAX_RETRANSMIT × RAND_FACTOR_fixed.

6.2 Verification setup:

we have used UPPAAL (model checker), with two templates composed over indexed channels (sand_c[id], sand_s[id]), plus a small queries file.

The parameters of CoAP [1]:
ACK_TIMEOUT = 2000 ms; RAND_FACTOR_fixed = 2; MAX_RETRANSMIT = 4; NSTART = 1;
D_ACK_MIN/D_ACK_MAX = 5/50 ms; small ID/token domains to keep the state space finite. The client resets its clock on every send and increments the retry counter only on retransmission.

Property	**Baseline (bounded service)**	**At boundary D_ACK_MAX = RTO_MAX**	**With small domains (wrap-around)**
ACK/CON matching (A[] !badAck)	✓	✓	✓
No token reuse while pending (A[] !tokenReuseWhilePending)	✓	✓	✓
One effect per MID (A[] duplicates_per_mid ≤ 1)	✓	✓	✓
Bounded retries (A[] k ≤ MAX_RETRANSMIT)	✓	✓	✓
NSTART discipline (A[] no overlap)	✓	✓	✓
Deadlock freedom (A[] not deadlock)	✓	✓	✓
Timing-liveness (A<> Done ∨ Abort) under D_ACK_MAX < RTO_MAX	✓	—	✓
Timing-liveness when D_ACK_MAX = RTO_MAX	—	✗ (counterexample)	—

Table 7: summary results

To run our model : Load the combined model in UPPAAL; confirm constants; run the safety, flow-control, deadlock, and liveness queries. For a boundary test, set D_ACK_MAX = RTO_MAX and re-run liveness to obtain the counter example trace. Table 6 gives a summary our results:

Most property are satisfied under the stated regime. Just the last one which indicates a violation and a the precondition of the property does not hold or is not applicable (e.g., liveness guaranteed only when D_ACK_MAX < RTO_MAX).

The safety and deadlock results establish that a standards-conforming endpoint must never acknowledges out of thin air, never reuses tokens while a request is pending, never applies multiple effects for the same MID, and never gets stuck. The timing-liveness result characterizes a simple, implementable bound—"server processing delay must be strictly smaller than the final retransmission window"—under which every exchange either completes or aborts predictably. This provides a clear, actionable rule for implementers configuring timeouts and maximum retries on constrained nodes.

6.3 Results and discussion

Safety holds under the baseline: every ACK corresponds to a prior CON, retries never exceed the configured cap, the NSTART rule is enforced, and the composition is deadlock-free. Under the timing assumption DACK_MAX<RTOmax, progress also holds: each exchange terminates—either by receiving a piggyback response or an empty-then-separate response, or by aborting cleanly after the final window.

At the exact boundary DACK_MAX=RTOmax, the model produces a counterexample (Fig. X) where the final ACK coincides with the last timeout, so the client aborts.

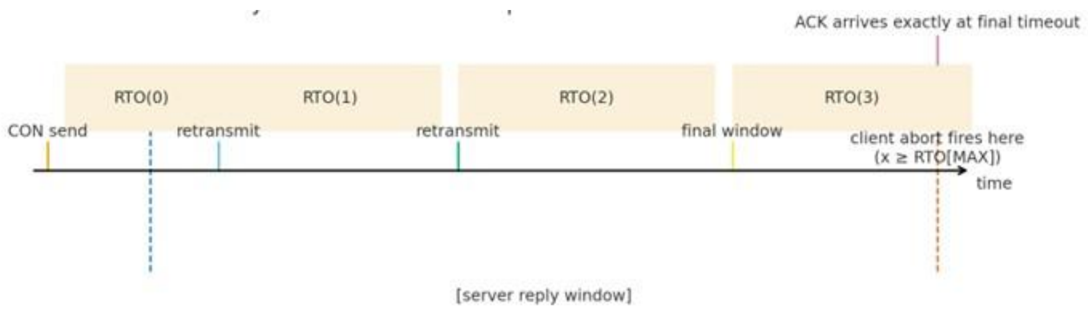


Figure 8: server replay

Liveness fails for a simple reason: the last ACK can arrive exactly at the client's final timeout, so the client gives up and the late ACK is ignored. This sharp threshold is useful in practice—it tells implementers that a strict inequality is enough to guarantee progress.

CONCLUSION

The promise of the Internet of Things (IoT) comes with a hard requirement: protocols must remain reliable on lossy links and tiny power budgets. CoAP has emerged as a central application-layer protocol, but its timing and control behavior must be understood and validated to ensure suitability for safety- and mission-critical deployments. In this work, we develop a compact, executable CoAP model in UPPAAL, validate core properties (ACK/CON matching, bounded retries, deadlock freedom), and make progress explicit by composing with a minimal network stub.

Beyond these core results, the model is intentionally simple and modular. We separate message mechanics from request–response correlation (Tokens), which keeps the automata small, auditable, and easy to extend. The network stub adds only what matters for realism—bounded one-way delay and an optional single duplicate—so progress reduces to a clear timing rule.

The present work addresses the creation of a formal CoAP model in UPPAAL and its validation as first step. Our claims are made under clear assumptions (no permanent loss for liveness, bounded integer domains, and a network abstraction with delay bounds and an optional single duplicate). The model already points to concrete next steps: add a small server-side deduplication cache to prove "one effect per MID" under duplication; introduce bounded server processing; and employ UPPAAL SMC for quantitative results (deadline-bounded reliability, latency percentiles, traffic/retransmission cost, and lightweight energy proxies). We also plan to scale the model to multiple clients and higher request rates to study fairness and NSTART interactions, and to exercise richer network conditions (loss, and reordering) while deepening the energy analysis. These extensions will not only broaden evaluation (scalability, reliability, energy) but can also inform practical enhancements and configuration guidance for CoAP in highly constrained deployments.

In summary, our contribution lies in building a formal foundation for CoAP validation. This foundation not only supports rigorous verification today but also offers a flexible basis for future studies that compare tools, extend to other IoT protocols, and ultimately guide the evolution of communication standards in the IoT ecosystem.

REFRENCES

[1] Shelby, Z., K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). 2014. RFC 7252. Internet Engineering Task Force (IETF). https://doi.org/10.17487/RFC7252
[2] Behrmann, Gerd, Alexandre David, and Kim G. Larsen. A Tutorial on UPPAAL. 2004. International Journal on Software Tools for Technology Transfer (STTT). Available: https://uppaal.org/texts/21-tutorial.pdf (Springer DOI: https://doi.org/10.1007/978-3-540-30080-9_7)
[3] Hartke, K. Observing Resources in the Constrained Application Protocol (CoAP). 2015. RFC 7641. IETF. https://doi.org/10.17487/RFC7641
[4] Bormann, C., and Z. Shelby. Block-Wise Transfers in the Constrained Application Protocol (CoAP). 2016. RFC 7959. IETF. https://doi.org/10.17487/RFC7959

[5] Bormann, C. Constrained Application Protocol (CoAP) Hop-Limit Option. 2020. RFC 8768. IETF. https://doi.org/10.17487/RFC8768
[6] Internet Engineering Task Force (IETF). The Internet of Things. 2024. https://www.ietf.org/topics/iot/
[7] Agarwal, A., S. S. Sonavane, and A. R. Pais. Modelling and Verification of CoAP over Routing Layer Using SPIN Model Checker. 2016. Procedia Computer Science 85: 816–823. https://www.sciencedirect.com/science/article/pii/S1877050916314557
[8] Chen, S., X. Li, and H. Zhu. Formalization and Verification of Group Communication CoAP Using CSP. 2022. In PDCAT 2021 (LNCS 13148), 698–709. https://link.springer.com/chapter/10.1007/978-3-030-96772-7_58
[9] Chen, S., H. Zhu, and Y.-F. Yuan. Formalization and Verification of Enhanced Group Communication CoAP. 2023. International Journal of Software Engineering and Knowledge Engineering 33(10): 1301–1327. https://worldscientific.com/doi/full/10.1142/S0218194023500535
[10] Larsen, K. G., A. Legay, M. Mikučionis, and D. B. Poulsen. UPPAAL SMC Tutorial. 2015. STTT 17(4): 397–415. https://link.springer.com/article/10.1007/s10009-014-0361-y
[11] Chen, Ningning, and Huibiao Zhu. IoT Modeling and Verification: From the CaIT Calculus to UPPAAL. 2023. IEICE Transactions on Information and Systems E106-D(9): 1507–1518. https://doi.org/10.1587/transinf.2022EDP7223
[12] Vogel, T., M. Carwehl, G. N. Rodrigues, and L. Grunske. A Property Specification Pattern Catalog for Real-Time System Verification with UPPAAL. 2022. Journal of Systems and Software 193: 111394. https://www.sciencedirect.com/science/article/pii/S0950584922002099
[13] Wang, Q., Y. Li, J. Zhang, X. Luo, and T. Wei. MPInspector: A Systematic and Automatic Approach for Evaluating the Security of IoT Messaging Protocols. 2021. In USENIX Security Symposium, 2367–2384. https://www.usenix.org/conference/usenixsecurity21/presentation/wang-qinying
[14] Gündoğan, C., P. Kietzmann, M. Lenders, H. Petersen, T. C. Schmidt, and M. Wählisch. NDN, CoAP, and MQTT: A Comparative Measurement Study in the IoT. 2018. arXiv:1806.01444. https://arxiv.org/abs/1806.01444
[15] Ebleme, M. A., C. Bayılmış, Ü. Çavuşoğlu, and K. Küçük. CoAP and Its Performance Evaluation. 2020. Sakarya University Journal of Science 24(1): 78–85. https://doi.org/10.16984/saufenbilder.613202
[16] Bansal, M., and Priya. Performance Comparison of MQTT and CoAP Protocols in Different Simulation Environments. 2020. In Inventive Communication and Computational Technologies, LNNS 145, Springer, 549–560. https://link.springer.com/chapter/10.1007/978-981-15-7345-3_47
[17] Silva, J., A. Carvalho, J. Soares, and R. Sofia. A Performance Analysis of IoT Networking Protocols: Evaluating MQTT, CoAP, OPC UA. 2021. Applied Sciences 11(11): 4879. https://doi.org/10.3390/app11114879
[18] Seoane, V., A. M. Alberti, L. Militano, and A. Iera. Performance Evaluation of CoAP and MQTT with Security Support for IoT Communications. 2021. Computer Networks 197: 108338. https://www.sciencedirect.com/science/article/pii/S1389128621003364
[19] Andersen, B., and T. Dalsgaard. Evaluating CoAP, OSCORE, DTLS and HTTPS for Secure Device Communication. 2023. In LNICST (EAI). (Publisher page as available)
[20] Palombini, C., M. Tiloca, and F. Palombini. Evaluating the Performance of the OSCORE Security Protocol in the Internet of Things. 2021. Internet of Things 14: 100366. https://www.sciencedirect.com/journal/internet-of-things/vol/14/suppl/C
[21] Selander, G., J. Mattsson, F. Palombini, and L. Seitz. Object Security for Constrained RESTful Environments (OSCORE). 2019. RFC 8613. IETF. https://doi.org/10.17487/RFC8613
[22] Chen, Ningning, and Huibiao Zhu. *IoT Modeling and Verification: From the CaIT Calculus to UPPAAL*. 2023. *Transactions on Information and Systems* E106.D (9): 1507–1518. https://doi.org/10.1587/transinf.2022EDP7223.
[23] Vogel, Thomas, Marc Carwehl, Genaína Nunes Rodrigues, and Lars Grunske. *A Property Specification Pattern Catalog for Real-Time System Verification with UPPAAL*. 2022. arXiv preprint arXiv:2211.03817. https://doi.org/10.48550/arXiv.2211.03817.