A Protocol for Autonomous, Device Status-Based Rollouts in Distributed Systems

Minsoo Kim, Taesoo Jun
Pervasive Intelligent Computing Laboratory, Department of Software Engineering
,Kumoh National Institute of Technology, Gumi, South Korea
{kms991022, taesoo.jun}@kumoh.ac.kr

***Abstract*—On July 19, 2024, a single flawed CrowdStrike content patch sent ripples of disruption across the globe, affecting an estimated 8.5 million Windows devices. Post-incident analyses underscored the need for phased deployments and rollback trials, yet they revealed a more fundamental vulnerability: a lack of end-to-end, protocol-level automated safety nets. This paper introduces Lightweight Metadata-based Synchronization, a mechanism designed to answer this challenge by embedding coordination policies directly into update manifests. This approach enables autonomous staged rollouts, triggers status-based automatic rollbacks, and deploys compatibility strategies to avert network fragmentation. The protocol's peer-to-peer status consensus forges resilience against partial network disruptions while upholding robust safety guarantees.**

***Index Terms*—Firmware Updates, Update Safety, Protocol Coordination, Distributed Systems, SUIT, Backward Compatibility**

I. Introduction

The paralysis of 8.5 million Windows devices on July 19, 2024, began with a single faulty content update [1]. While CrowdStrike's subsequent review correctly highlighted the need for more robust, staggered rollouts [2], the event signaled a deeper truth: operational *processes* for safety existed, but they lacked the teeth of *protocol-enforced* automatic guardrails.

This event was a symptom of a deeper vulnerability lurking within any large-scale distributed system, from industrial IoT and edge computing to CDNs and container platforms. Without coordination baked into the protocol, version fragmentation can splinter a network, isolating devices and disrupting services. While established frameworks like SUIT [3] and TUF/Uptane [8], [9] secure the delivery pipeline, they delegate the complex task of coordination to higher-level software.

Contemporary deployment systems reveal critical gaps. Kubernetes rolling updates offer a lever to pause a troubled rollout, but yanking it back requires manual intervention via `kubectl rollout undo` or an external controller [4]. AWS CodeDeploy can trigger rollbacks from alarms, but this depends on manually configured CloudWatch metrics and does not verify compatibility between coexisting versions [5]. The proposed protocol shifts the burden of safety from external monitors and human operators to an autonomous policy embedded within the update's fabric.

II. Protocol Design

The protocol's intelligence is not centralized but distributed, embedded directly into the update metadata of each device (Fig. 1).

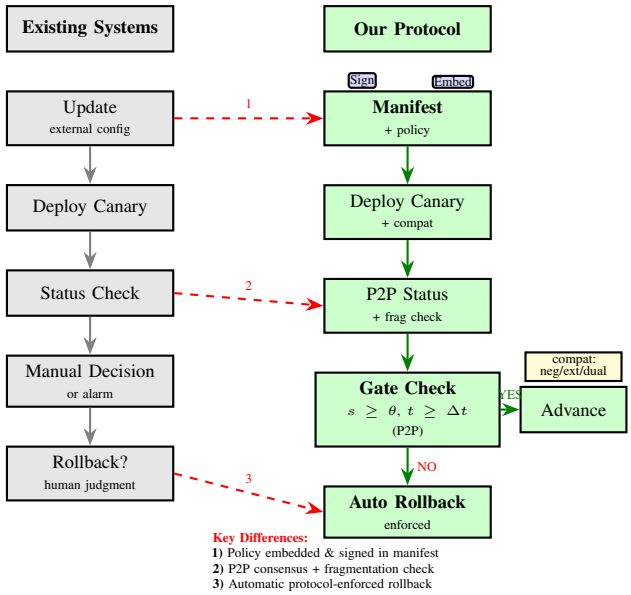

Fig. 1. Protocol-enforced rollout: existing systems rely on external configuration and manual intervention (left), while our protocol embeds signed policy in manifests for autonomous coordination (right). **Novelty:** policy carried in signed manifest; gates use decentralized P2P consensus with inter-version probes; compatibility strategy enforced during coexistence.

A. Embedded Policy Mechanism

At its heart lies a compact, CBOR-encoded policy[1] inside the update manifest. This is not static data; it is a signed, executable set of rules. The policy dictates the entire deployment choreography: defining stages (e.g., a 1–5% canary cohort), setting conditions for advancement (e.g., operational metrics stable above 95% for two hours), and establishing triggers for an automatic rollback. It also configures the peer-to-peer reporting mechanism, which relies on cryptographic signatures to ensure authenticity.

B. A Spectrum of Compatibility Strategies

To keep the network from fracturing into incompatible cliques during a rollout, the `compat_strategy` field specifies one of three approaches. The choice hinges on the update's nature—a breaking change versus an additive feature—along with device resource constraints and transition time tolerance.

Version Negotiation: Devices advertise their supported versions, communicating with the highest one they share. Its

[1]CBOR (RFC 8949) is a binary data serialization format optimized for small code and message sizes [6].

minimal overhead (<100 bytes) makes this strategy ideal for resource-constrained devices. **Backward-Compatible Extensions:** New capabilities are introduced as optional, version-tagged extensions, which lets older devices safely ignore what they do not understand while maintaining core functions. **Dual-Stack:** This approach gives the strongest safety guarantee for breaking changes at the cost of significant memory overhead, most suitable for major version shifts in well-resourced gateways.

C. Operational Flow

The deployment process follows a carefully orchestrated sequence, starting with the broadcast of the signed manifest containing the embedded policy. Each device then uses this policy to self-organize and activate the specified compatibility strategy. Nodes engage in peer-to-peer verification[2], cross-checking stability metrics and ensuring inter-version communication remains viable. Any detected status degradation or fragmentation prompts an immediate, autonomous halt and rollback.

III. Security Considerations

A peer-to-peer status monitoring model inevitably creates attack vectors. An adversary could inject false reports to stall a critical update or, conversely, rush a malicious one. The protocol's design integrates several mitigation strategies against these threats. Signed beacons help ensure message authenticity, while aggregation methods robust against outliers, such as a trimmed mean or median, prevent malicious nodes from skewing the consensus.

Gossip-based dissemination provides both efficiency and redundancy. In cases of conflicting reports, a "freeze-on-conflict" behavior prioritizes system safety. This decentralized architecture is inherently resilient, allowing for graceful degradation in edge and IoT settings where a central coordinator might become unreachable.

IV. Impact Analysis

Had this protocol been active during the CrowdStrike incident, a policy mandating a 1% canary cohort would have immediately capped the impact to roughly 85,000 devices, not 8.5 million. The moment operational metrics plunged or fragmentation signals appeared, an autonomous reversion would have triggered without human intervention, manual alarm configuration, or arbitrary timeouts. While the affected canary group rolled back, the vast majority of devices would have continued operating normally.

Moreover, the enforced compatibility strategy would have kept communication channels stable between versions, preventing network partitioning during the rollback window. Unlike process-based controls that an operator might circumvent, these embedded policies are cryptographically signed and verified by each device.

This mechanism complements existing frameworks. While SUIT [3] ensures secure firmware delivery and TUF/Uptane [8], [9] tackle supply chain security, neither dictates the terms of autonomous, status-based coordination or enforces inter-version compatibility during staged deployments.

[2]Gossip-based averaging algorithms are known for their rapid (exponential) convergence, with the rate depending on the network graph's spectral properties [7].

V. Conclusion and Future Work

The 2024 CrowdStrike event serves as a stark reminder that staged rollout processes remain fragile without protocol-enforced guardrails. The Lightweight Metadata-based Synchronization protocol addresses this gap by embedding coordination policies and compatibility strategies into update manifests, offering a new paradigm for resilient deployments with autonomous staging, status-verified progression, and automatic rollbacks while preventing network fragmentation.

Future work will focus on implementing a prototype on resource-constrained IoT devices and conducting large-scale simulations (1,000–10,000 nodes) to validate convergence properties under network partitions. We plan to measure protocol overhead, evaluate rollback latency against manual intervention baselines, and apply the protocol to historical incident data to quantify its potential for blast radius reduction.

Acknowledgment

This research was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Ministry of Science and ICT (MSIT) through the Innovative Human Resource Development for Local Intellectualization program (IITP-2025-RS-2020-II201612; 34%) and the Information Technology Research Center (ITRC) program (IITP-2025-RS-2024-00438430; 33%); and by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Education through the Basic Science Research Program (2018R1A6A1A03024003; 33%).

References

[1] Microsoft, "Helping our customers through the CrowdStrike outage," Azure Blog, Jul. 2024.
[2] CrowdStrike, "External Technical Root Cause Analysis - Channel File 291," Aug. 2024.
[3] B. Moran et al., "A Firmware Update Architecture for Internet of Things," RFC 9124, Jan. 2022.
[4] Kubernetes, "Kubernetes Deployments: Rolling Update," Kubernetes Documentation, 2024. [Accessed: Oct. 2025]
[5] Amazon Web Services, "Redeploy and Roll Back a Deployment with CodeDeploy," AWS Documentation, 2024. [Accessed: Oct. 2025]
[6] C. Bormann and P. Hoffman, "Concise Binary Object Representation (CBOR)," RFC 8949, Dec. 2020.
[7] S. Boyd et al., "Randomized Gossip Algorithms," *IEEE Trans. Information Theory*, vol. 52, no. 6, pp. 2508–2530, Jun. 2006.
[8] J. Cappos et al., "A look in the mirror: Attacks on package managers," *Proc. ACM CCS*, pp. 565–574, 2008.
[9] Uptane Alliance, "Uptane IEEE-ISTO Standard for Design and Implementation," IEEE-ISTO 6100.1.0.0, Jul. 2019.