HASP - High Availability Standards Proxy

andrew@techonite.com

March 30, 2026

Abstract

This document challenges the effectiveness of a client-server architecture[1] with regard to schema application, evolution, and communication between two or more parties over a computer network. An alternative architecture, HASP (High Availability Standards Proxy), is proposed as a solution to enhance reliability via a relay, or proxy-based approach.

Fundamentally, a HASP is a schema registry that sits outside of a client and server's network boundaries, and through which messages are relayed. The HASP's availability for both (or all) parties through the internet opens up interesting possibilities for what is referred to in this document as *schema arbitration*; namely, the ability to negotiate, validate, version, inform, and evolve shared data structures through the facilitation of a third party.

The term *hasp* is used colloquially to mean one part of a two-part locking system - the hasp itself, generally a hinged plate, and a *staple*, a ring over which the hasp is secured. A lock may then be used to secure the hasp to the staple.

This three-part system is a convenient metaphor for the proposed architecture, in which a client and server's communications are bound and managed through an additional proxy component.

Contents

1 Introduction **1**

2 Methodology **2**
2.1 Arbitration 2
2.2 Communication 3
2.3 Traffic Modes 3
2.4 Deployment Modes 4
2.5 Locking Mode 4
2.6 Availability 5
2.7 Clients 5
2.8 Inversion of Control 5
2.9 Agentic Applications 6
2.10 Tradeoffs 6

3 Conclusion **6**

1 Introduction

Integration and *communication*, not operation (algorithms), are among the most critical functions, and challenges of, modern software systems. More specifically, it is the inability of two

[1]The HASP model still presupposes ***requests are initiated by the client*** [1]. It differs insofar as it does not assume the origin server is the preferred candidate for producing a response.

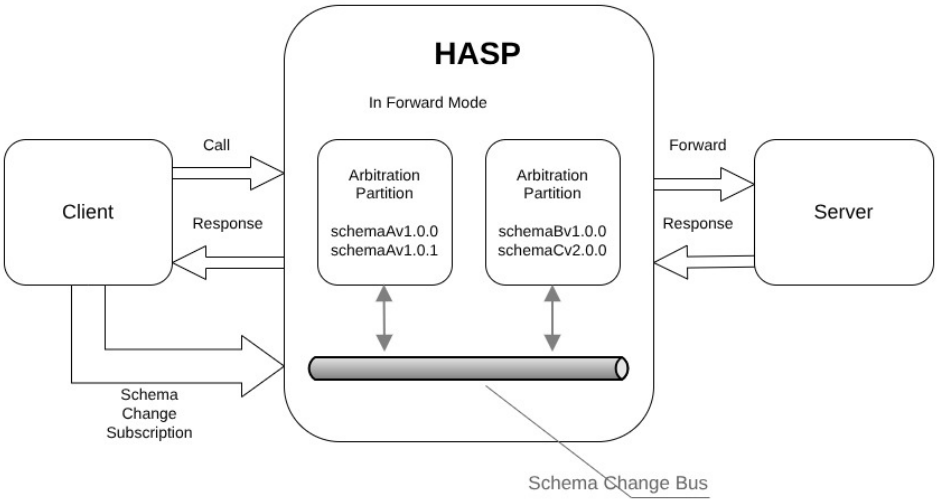

Figure 1: Fundamental HASP Architecture - Forward Mode

or more parties to effectively negotiate a shared understanding of data structures that leads to many of the most common and disruptive failures in distributed systems[2].

This document, and the architecture proposed therein, is a result of direct experience with internationally maintained software standards and the challenges that stem from their implementation as the standards evolve. A non-exhaustive list of such challenges includes:

- Difficulties in identifying one schema among two or more possible candidates maintained in version control or API documentation
- Vetting evolving data structures for inclusion in schema-on-read systems
- The application of stale schemas in validator applications
- Differences that occur between implementations of the same validation standard
- A lack of clarity regarding what constitutes a breaking change
- Difficulty in coordinating data interchange formats between systems
- An inability to effectively aggregate usage metrics for shared schemas

It should be noted that such challenges are not unique to publicly available software standards, but may be commonplace in any system with evolving data structures - even between development teams within the same organization. Moreover, HASP's services are not new, as offerings exist to help developers navigate schema evolution and validation. What makes this architecture effective is *where* these services are placed in relation to the communicating parties, and how they are applied.

2 Methodology

2.1 Arbitration

A HASP is a proxy server that exposes an arbitration partition (a workspace or tenancy) for two or more parties to negotiate the data structures that will pass between them. This may be

[2]This document leverages the **Confluent** definition [2] of distributed systems as "a collection of independent components and machines located on different systems, communicating in order to operate as a single unit."

done via a Graphical User Interface (GUI), or Application Programming Interface (API). Once a data structure is agreed upon, an immutable copy of that schema is stored in the partition's schema registry. New versions of a schema do not subsume or invalidate previous versions, and many versions may be active at any given time.

All schemas in the registry are versioned using **semantic versioning** [3]. While the proxy may allow an initial version to be set manually, derived versions are generated automatically based on the Evolution Rule Set (ERS), a set of criteria that define how changes to a schema affect its version number. This setting may not be overridden without appropriate authorization. Automated versioning is crucial for ensuring that applications can efficiently determine whether in-flight messages conform to consumers' expectations. It also prevents the potential for standards bodies to undermine the scope of their changes in order to facilitate adoption.

Data structures stored in the registry may be defined using any number of currently supported schema definition languages, including **JSON Schema** [4], **XML Schema (XSD)** [5], **Apache Avro** [6], **Protocol Buffers** [7], and others. As messages pass into (or through) the HASP, they are validated against the appropriate schema version. Additionally, in-flight messages may be:

- Re-serialized using a different data interchange format
- Inspected for the presence of fields aligning with privacy regulations
- Analyzed for usage metrics

Importantly, the semantic content of proxied message bodies may not be altered by or stored on the proxy server, though HTTP headers may be supplied to inform downstream parties [3].

2.2 Communication

In many contemporary software systems, software artifacts (compiled, interpreted, or otherwise) directly or indirectly drive the production of software specifications. As such, the specifications often lag behind the code, or may be incorrect entirely. The drift in understanding between what data a system should accept and what it actually accepts can and does result in lost revenue due to time and friction between coordinating teams. Note that this experience may be as frustrating for business analysts and customer success associates as it is for developers, requiring person-to-person communication with external vendors for clarification on supported APIs.

A HASP architecture inverts this model. When messages that pass into or through the proxy server *must* conform to the published contract to reach the consuming application, then the schema drives the creation of software artifacts to support the contract. This kind of contract enforcement becomes more critical as the number of communicating parties grows, as it does in public, standardized protocols.

To inform interested parties of changes to the standard, any alteration to a schema in the HASP's registry that results in automatic versioning of that schema[4] is broadcast to the Schema Change Bus, a publish-subscribe channel whose permissions are managed by members of the arbitration partition. Data broadcast on the Schema Change Bus may also be used to keep privately deployed validators in sync with the hosted proxy.

2.3 Traffic Modes

Proxies operate in two primary traffic modes, depending on the type of integration and the level of security required: *forward* and *redirect*. Traffic modes determine how messages flow between

[3] Messages always flow downstream [8, p. 44]

[4] Draft Schemas are not versioned until finalized

the client, proxy, and origin server[5]. They also determine the range of possible behaviors the proxy may exhibit.

Forward mode allows messages to pass through the proxy server, which increases the amount of control provided to the proxy. When in forward mode, the proxy may:

- Forward valid messages from the client to the origin server
- Reject or annotate[6] invalid messages from the client before they reach the origin server
- Return valid responses from the origin server to the client
- Annotate invalid responses from the origin server before they reach the client

Alternatively, redirect mode allows the client to maintain a direct connection with the origin server, which may be required in some scenarios.

In redirect mode, the proxy may only:

- Return a validation response to the client, indicating whether the request may be redirected to the origin server.

It is natural for both client and server teams to demand transparency around HTTP traffic flowing through the proxy. To that end, HASP proxies provide detailed metrics and visualizations which document traffic patterns, schema validation results, and other relevant information.

2.4 Deployment Modes

While managing a shared workspace for defining and validating data structures is one of the primary functions of this architecture, it is not strictly necessary for traffic to pass through the hosted proxy itself, if doing so violates security or compliance requirements for an organization[7].

Any time a schema is versioned in the workspace, an image of the validation code is built and published to a container registry. This may then be deployed within the client's or server's network perimeter for over-the-wire validation. This style of deployment is referred to as *private mode*. Privately deployed validators may be kept in sync with the hosted proxy via the Schema Change Bus, ensuring that data structures do not grow stale over time.

It should be noted that in-flight data that passes through the hosted proxy (public mode) offers significant advantages in terms of observability and control, and is the preferred method of operation for most use cases.

2.5 Locking Mode

[8] This paper makes the fundamental assumption that a significant number of program or system crashes can be attributed to misaligned or unexpected data structures, and that many of these data structures are introduced to systems at network boundaries, rather than produced by subroutines. Further, while defensive programming via exception handling or monadic computations will continue to remain best practice, there is always a degree to which these practices will be accidentally omitted or fail to perform as expected.

A HASP architecture assumes that there is *no good reason* that non-compliant data should reach its intended destination, and that the responsibility of validation and defense should be

[5] We define the origin server as the intended recipient of HTTP messages from a client, to whom any additional requests from the origin server to downstream servers are opaque.

[6] Annotation refers to the act of producing HTTP headers that describe how a message has failed to conform to its defined standard

[7] Consider SOC 2 [9]

[8] Locking mode is optional; it may be omitted in scenarios where the proving overhead is prohibitive.

the primary focus of the proxy, and not of the origin server, or of the client. The expected result of this assumption is a significant increase in the number of 4xx HTTP response codes and a decrease in the number of 5xx HTTP response codes[9].

In order to effect this defensive position, the proxy server may:

- Require the client to prove it can send messages that conform to data structures in the HASP's registry[10].
- Require the server to prove it can accept messages conforming to data structures in the HASP's registry[11].

Until both conditions are satisfied, valid messages from the client are returned to the client by the proxy, indicating that the system has not yet locked[12]. Only once this is the case will the proxy forward messages from the client to the origin server.

2.6 Availability

Today, high availability software systems achieve uptime (predominantly) through redundancy and failover, where redundant components are more or less identical to one another [13]. This paper argues that such systems, while both necessary and broadly effective, form a *redundancy monoculture*, in which systemic failures may occur due to shared vulnerabilities across components. Instead, the HASP architecture promotes a *sympathetic architecture*, where redundant components accomplish the same goals as the primary components, but do so in different ways and through different network channels (sometimes at the risk of a reduced feature set or operational efficiency).

To achieve maximum uptime, it is recommended to deploy the proxy in both public and private mode, reflecting an *active-passive* DR (Disaster Recovery) strategy. Using this model, messages are preferentially routed through the publicly available HASP server. If, however, a connection should be lost to the public proxy, traffic may fail over to the privately deployed container until connection to the SaaS solution is reestablished.

2.7 Clients

In order to route HTTP calls through the proxy server, teams can choose from a number of HASP client libraries which wrap existing HTTP solutions in several languages (Python, Java, Node, Go, etc.). Requests may manually specify the schema version (via headers) against which the HASP validates the request body. Preferably, however, the client library may connect with the arbitration partition's schema registry at runtime to autodetect the version being sent in the payload. This leaves very little work (or room for error) on the part of the client, and allows for more dynamic interactions with the proxy server.

2.8 Inversion of Control

As noted in Section 2.1, *the semantic content of proxied message bodies may not be altered by or stored on the proxy server...*. This reflects the expectation that clients should have of the proxy, which is that it in no way interferes with or changes the meaning of their data. With

[9]This expectation is based on architectural reasoning rather than empirical measurement; validation is left to future work.

[10]There is some assumption here about the HTTP method being leveraged

[11]This is accomplished by automatically generating and forwarding compliant traffic from the proxy to the origin server. Such traffic may be adorned with HTTP headers to indicate that it is a probing message.

[12]A HASP implementation may introduce its own response codes to communicate proxy-specific states to the client.

[13]Consider Apache Kafka replicas [10]

that being said, there is often a very real need to map one business's data structure to another business's data structure, and the proxy server is a logical place to address this need. As such, clients are offered the ability to deploy mapping or transformation code into their arbitration partition to be invoked by the HASP's IoC container. Clients may implement the mapping code in a variety of languages by overriding one or more virtual methods, or by injecting composable functions into the container. In the event IoC is leveraged, HASP validation against the schema registry will occur both before and after the mapping functions are applied.

2.9 Agentic Applications

This paper was written with the intention of aiding human software engineers in building more robust web architectures for in-flight data.

However, as details of the HASP architecture have emerged, it has become increasingly clear that such a system is equally well-suited as an agentic API negotiation platform, in which software agents representing the interests of their respective teams may *safely* negotiate data structures and facilitate communication between systems. Viewed through this lens, the proxy serves as the primary HIL (Human in the Loop) Control Center, allowing programmers and business stakeholders to review, approve, or decline data structures proposed by agents on their behalf.

Consider that an agent running on the proxy (the arbitration agent) may securely facilitate communication between agents running inside the client and server networks, acting on behalf of each team to discover and agree upon shared data structures. This would allow for rapid, automated exploration of each system's API surface and data expectations, resulting in a more efficient integration process.

2.10 Tradeoffs

There is no doubt that a HASP architecture introduces additional complexity and latency to the systems that leverage it, and is certainly not the right choice for every use case. The overhead of schema management is most justified when a small number of data structures are shared across a large number of teams, where the cost of governance is amortized across many integrations. By contrast, two teams managing a large number of distinct or rapidly evolving structures may find the overhead outweighs the benefit.

HASP is particularly well-suited for highly regulated industries, systems where latency is not a primary concern, or teams overcoming significant integration challenges, such as those arising from mergers and acquisitions.

3 Conclusion

At first glance, a HASP architecture may seem burdensome. Yet the many hours of schema interrogation, communication overhead, and system crashes that result from a contemporary web architecture leave a great deal to be desired.

This document argues that while the core architectural components of the web (clients, proxies, and servers) are sound, the way in which they are currently leveraged can be significantly improved upon. The introduction of a HASP architecture can enhance the reliability of those systems and drastically reduce friction between communicating parties. As the role of software agents in API negotiation continues to grow, the architecture's potential to serve as a neutral arbitration platform (keeping humans informed and in control) may prove to be among its most valuable characteristics.

References

[1] MDN Web Docs. An overview of HTTP, 2025. URL https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Overview. Accessed: 2026.

[2] Confluent. What are distributed systems?, 2024. URL https://www.confluent.io/learn/distributed-systems. Accessed: 2026.

[3] Tom Preston-Werner. Semantic versioning 2.0.0, 2013. URL https://semver.org/. Accessed: 2026.

[4] JSON Schema. JSON schema, 2020. URL https://json-schema.org/docs. Accessed: 2026.

[5] World Wide Web Consortium. XML schema, 2012. URL https://www.w3.org/XML/Schema. Accessed: 2026.

[6] Apache Software Foundation. Apache avro, 2024. URL https://avro.apache.org/docs/. Accessed: 2026.

[7] Google. Protocol buffers, 2024. URL https://protobuf.dev/overview/. Accessed: 2026.

[8] David Gourley et al. *HTTP: The Definitive Guide*. O'Reilly Media, 2002.

[9] Palo Alto Networks. SOC 2 — System and Organization Controls, 2024. URL https://www.paloaltonetworks.com/cyberpedia/soc-2. Accessed: 2026.

[10] Confluent. Kafka replication, 2024. URL https://docs.confluent.io/kafka/design/replication.html. Accessed: 2026.