Universal Library of Engineering Technology Research Article

ISSN: 3064-996X | Volume 3, Issue 1

Open Access | PP: 122-129

DOI: https://doi.org/10.70315/uloap.ulete.2026.0301021

A Telemetry-Driven Pattern for Preventing Stale Client States in Single-Page Applications: Architecture, Instrumentation, and Production Results

Evgenii Lvov

Head of Engineering, Senior Full Stack Architect, Batumi, Georgia.

Abstract

In the context of modern web development, the transition to a single-page application architecture has radically transformed the paradigm of working with state: responsibility for maintaining data consistency has shifted to a much greater extent from the server side to the client side. This redistribution of responsibilities has exposed a systemic problem of state drift, in which the information displayed in the user interface ceases to correspond to the canonical server-side source. Such discrepancies generate direct financial losses in e-commerce systems, lead to failures and inefficiencies in logistics platforms, and significantly degrade the quality of user interaction in collaborative applications. Within the framework of this study, the architectural pattern Telemetry-Driven State Invalidation (TDSI) is examined and analyzed in detail as a methodological approach to cache invalidation management based on real-time analysis of client state telemetry. In contrast to classical schemes of periodic polling or persistent connections with high resource costs, TDSI relies on observability tools (in particular, OpenTelemetry) to form an adaptive control loop that triggers data updates only when semantically significant divergences are detected. Based on the analysis of real-world production scenarios (Uber, Slack, DoorDash) and comparative benchmarks of transport protocols, it is shown that the use of telemetry-driven invalidation makes it possible to reduce network load by up to 40% while preserving data freshness metrics (Age of Information) comparable to push-oriented models, and also demonstrates a substantial advantage over traditional TTL-based strategies in terms of state accuracy and energy efficiency. The materials of the study will be of particular interest to frontend architects and team leads, distributed systems engineers, observability/telemetry specialists, and developers of high-load SPAs in e-commerce, logistics, and collaborative platforms.

Keywords: *Single-Page Applications, Client State Drift and Stale Caches, Telemetry-Driven State Invalidation, Data Freshness Metric Age of Information, Observability Tools, Server-Sent Events.*

INTRODUCTION

Historically, the prevailing model for building web applications was the Multi-Page Application (MPA) architecture, within which the lifecycle of state strictly coincided with the lifecycle of an HTTP request [1]. The industry's transition to Single-Page Applications (SPA) and the widespread adoption of frameworks such as React, Angular, and Vue have radically changed this architectural paradigm [3]. The browser has evolved into a client that maintains a long-lived session and local state capable of persisting for hours, and in certain scenarios even days, without a page reload. As a result, the frontend has essentially become a distributed system in which each client instance acts as an asynchronous replica of the server data state. However, unlike server replicas, such client copies operate in a fundamentally unreliable environment of the open internet, possess limited resources (CPU, memory, battery) and, importantly, lack built-in consensus mechanisms that guarantee strict consistency among participants of a distributed system [4].

The phenomenon of state drift is understood as a situation in which the local copy of data on the client begins to diverge from the canonical server state, the single source of truth [6]. The problem of state drift becomes particularly critical in systems with highly dynamic data. In e-commerce, the average accuracy level of inventory data is only about 83%, which directly leads to order cancellations, increased returns, and reduced user loyalty [8]. Under Flash Sale conditions, even a minor delay in updating an item's status, on the order of several seconds, can provoke overselling and cause noticeable reputational costs. In logistics and delivery,

Citation: Evgenii Lvov, "A Telemetry-Driven Pattern for Preventing Stale Client States in Single-Page Applications: Architecture, Instrumentation, and Production Results", Universal Library of Engineering Technology, 2026; 3(1): 122-129. DOI: https://doi.org/10.70315/uloap.ulete.2026.0301021.

for services such as Uber or DoorDash, discrepancies in the coordinates of the contractor or in the order status are practically unacceptable: the speed of data is part of the very consumer value of the product, and any staleness of state is immediately reflected in operational efficiency and user experience [10]. In collaborative systems such as Slack or Google Docs, users expect quasi-real-time reflection of their colleagues' actions; desynchronization of the client cache leads to working with an outdated context, duplication of operations, the emergence of conflicting versions and, as a result, a loss of trust in the system as a means of joint activity [12].

These effects are amplified by the widespread use of the Optimistic UI pattern. To compensate for network latency and improve the subjective quality of UX, the interface is updated before receiving confirmation from the server, creating the impression of an instantaneous response. Empirical studies show that such an immediate reaction reduces the perceived waiting time by approximately 40% [14]. However, in the event of failures in background synchronization or concurrent data modifications on the server, the interface may enter a state of a kind of hallucination, visualizing for the user a reality that no longer exists and has already been lost [15].

At the same time, the academic and engineering literature does not sufficiently elaborate hybrid, adaptive strategies that would rely on observability signals and dynamically adjust the synchronization strategy to the actual behavior of the system and users.

The aim of the study is to conceptualize, formalize, and substantiate the Telemetry-Driven State Invalidation (TDSI) pattern. This approach is understood as the use of telemetry data (based on OpenTelemetry) to close the control loop over state: the client periodically publishes metadata about its local state, and the server, analyzing this telemetry, decides on the advisability and scope of invalidation.

The hypothesis is put forward that telemetry-driven invalidation of client state (TDSI), implemented as a closed control loop based on OpenTelemetry and SSE, makes it possible to simultaneously radically reduce network and energy load while maintaining data freshness at a level comparable to push-oriented systems.

The scientific novelty lies in the formalization of the presented pattern for SPA through the Age of Information metric and the principles of control theory, its comparison with classical TTL/polling/WebSocket approaches, and the generalization of production cases (Uber, Slack, DoorDash) into a reproducible architectural template.

MATERIALS AND METHODS

As the empirical basis of the study, a combination of academic and industrial sources was used, with an emphasis on works describing consistency in distributed systems and practices for building SPA. The academic corpus included publications on client-side replication and weak consistency models, studies on strong consistency in distributed data stores, works on CRDT and their practical implementations (Yjs, Automerge), as well as articles introducing and developing the Age of Information metric and methods for its optimization in sensor and IoT networks. In addition, studies on WebSocket/SSE infrastructure and real-time web (including performance benchmarks) were taken into account, as well as works on UX effects of latency and optimistic interfaces. The search was conducted in the IEEE Xplore and ACM Digital Library databases and in specialized web engineering journals using the following keywords: single-page application, client-side replication, Age of Information, WebSockets, Server-Sent Events, CRDT, observability, closed-loop control.

A separate block of materials consisted of industry reports and analytics on the modern frontend landscape, which were used to contextualize the significance of the state drift problem and to assess the prevalence of SPA architectures. This group included surveys and reports such as State of Frontend 2024, as well as articles and blog posts devoted to the demand for SPA in 2024–2025 and the evolution of client-side architecture. The inclusion criterion for such sources was the presence of quantitative assessments (share of SPA, prevalence of real-time features, protocols used) and an explicit link to real production systems rather than to demonstration or educational examples. Information from marketing and promotional materials was not included if it did not contain reproducible metrics or a clear description of the technical stack.

All collected data were subjected to qualitative and quantitative synthesis: academic sources defined the theoretical framework (consistency models, AoI, closed-loop control), industrial cases defined the boundaries of solution feasibility and typical architectural patterns, and e-commerce statistics defined business constraints and target metrics (conversion, inventory accuracy, share of technical failures). At the intersection of these three layers, the requirements for the TDSI pattern were formulated: acceptable level of AoI, limits of network and resource load, required properties of the transport layer (unidirectional streaming, automatic recovery, friendliness to proxies and firewalls). These requirements were then used to construct the architectural scheme of TDSI and its parameters (structure of the telemetry vector, drift detection algorithms, dynamic TTL, interaction with CDN), as well as to interpret the numerical estimates of traffic and consistency gains reported in the study.

RESULTS AND DISCUSSION

Considering an SPA as an independent node of a distributed system inevitably requires applying the framework of the CAP theorem (Consistency, Availability, Partition Tolerance) to it. Under conditions of unstable mobile internet, the assumption of the presence of a network partition ceases to be an

abstraction and becomes an objective physical reality, so the developer is effectively forced to choose between preserving strict consistency and maintaining high availability [4]. An additional refinement is provided by the PACELC theorem: even in normal operation mode, when there is no network partition (Else), the system faces a fundamental trade-off between latency and consistency. For a user interface, significant latency is fundamentally unacceptable, as it directly correlates with conversion, retention, and audience engagement metrics [17, 18]. As a result, the overwhelming majority of SPA-based decentralized frontends implement eventual consistency by default, sacrificing immediate strong consistency in favor of the perceived speed of the interface. However, the horizon of this eventual consistency may turn out to be practically unacceptable from the standpoint of user experience and business metrics. Research in the field of client-side replication [4] shows that under such conditions causal consistency is the strongest realistically achievable consistency model, but its implementation on the client side requires a nontrivial mechanism for tracking causal dependencies between operations and versioning of states, which significantly complicates the architecture and increases maintenance costs.

One of the strictly formalized approaches to managing replica divergence is the class of data structures known as CRDT (Conflict-Free Replicated Data Types). These mathematically designed data types, implemented in practice in libraries such as Yjs and Automerge, provide guaranteed convergence: any concurrent changes on different nodes can be merged without logical conflicts, so that all replicas eventually converge to a single consistent state. Nevertheless, CRDTs solve only the problem of reconciliation and merging of updates that have already been delivered (for example, in collaborative text editing scenarios), without addressing the fundamental transport-level problem of timely and reliable delivery of these updates between participants in the system. In addition, the CRDT approach is often accompanied by significant metadata overhead and an increase in the volume of transmitted and stored information (the metadata explosion effect), which makes such solutions excessive for typical CRUD interfaces and economically inefficient in terms of performance and energy consumption, especially on mobile clients.

For a quantitative description of the phenomenon of client state aging in this work the Age of Information (AoI) metric is used, originating from control theory and sensor network research. In contrast to classical latency, which measures the transmission time of an individual packet from source to receiver, AoI measures the time elapsed since the generation of the latest update that has already reached the client. Formally, let u(t) be the timestamp of the update that is present at the client at time t. Then the instantaneous age of information Δ(t) is defined as:

$$\Delta(t) = t - u(t) \quad (1),$$

where: the graph of Δ(t) is a sawtooth function: in time intervals without updates the age of information grows linearly, whereas upon arrival of a new update an instantaneous reset-like drop of Δ(t) occurs. The key task of the TDSI pattern is formulated as minimization of Peak AoI (PAoI), that is, the maximum value of the age of information, while simultaneously observing constraints on the communication channel bandwidth B.

Research in the areas of IoT and sensor networks indicates that the optimal state update strategy does not always coincide with the naive zero-latency policy; on the contrary, an effective strategy is often to maintain AoI below a certain threshold value that depends on the semantics and criticality of the specific data. This theoretical justification underlies the rejection of constant continuous push (WebSockets as a continuous stream) in favor of a more economical, controlled state invalidation.

The TDSI concept is based on the principles of automatic control theory in closed-loop systems. In terms of the classical control scheme, the controlled object is the state of the client application, primarily its cached data. The role of the sensor is played by telemetry collected, in particular, by OpenTelemetry and containing information about the current version of the data on the client and user activity parameters. The controller is implemented as a server-side Drift Detection component that compares the client version with the canonical server version and estimates the magnitude of the drift. The actuator is represented by an invalidation signal that triggers an update of the local state on the client. The use of such feedback makes the system adaptive: the frequency and volume of updates are adjusted dynamically depending on the actual user behavior and data volatility. According to existing studies, such closed-loop control schemes provide higher stability and resource efficiency compared to static, predefined update schedules, where synchronization parameters are fixed and do not respond to the current context [2, 5].

The choice of the mechanism for delivering invalidation signals in the TDSI architecture is a fundamentally important system design decision. In this work, three basic approaches are considered and compared: classical HTTP polling, its long-polling modification, as well as the Server-Sent Events (SSE) and WebSockets protocols, with the conclusions being based not only on theoretical considerations but also on empirical performance benchmark data.

In the case of periodic HTTP polling, the client initiates a request to the server at fixed time intervals equal to N seconds. This approach has obvious advantages: it is extremely simple to implement, relies on a stateless architecture on top of standard HTTP and, as a consequence, is almost transparent to the existing network infrastructure such as proxies,

load balancers, and corporate firewalls. At the same time, the fundamental limitations of such a mechanism are well known. The average event delivery latency under uniform polling is on the order of N/2, which is unacceptable for many interactive scenarios. In addition, a significant portion of requests carries no new information while still creating load on the server and the communication channel. Each such poll, if aggressive use of keep-alive connections is not employed, is accompanied by the establishment of a TCP/TLS session and the transmission of the full set of HTTP headers and cookies, which leads to substantial protocol overhead, especially noticeable on mobile networks [17].

The long-polling modification attempts to partially compensate for these drawbacks: the connection is kept open until new data appear, after which the server immediately returns a response and the client re-initiates the request. This does indeed reduce the average event delivery latency, but it is achieved at the cost of increased resource consumption on the server side. In a synchronous implementation, threads or workers become blocked, and when scaling without the use of non-blocking I/O this leads to rapid exhaustion of server resources. Additionally, under conditions of unstable connectivity and connection breaks, issues with preserving message delivery order may arise, which complicates the logic of state recovery on the client.

WebSockets represent a full-duplex protocol on top of TCP, providing a persistent bidirectional data exchange channel between client and server. From a performance standpoint, this approach ensures minimally possible latency and very high throughput: benchmarks demonstrate the capability to handle up to several million events per second with batched message transmission. However, these advantages are accompanied by significant systemic costs. First, there is the scalability problem: maintaining millions of simultaneously open TCP connections requires a specialized architecture, fine-tuning of operating system kernel parameters, and careful management of file descriptors, which become the limiting resource. Second, on mobile devices, continuously maintaining an active socket and exchanging keep-alive packets prevents the radio module from entering power-saving idle states, which leads to a noticeable increase in energy consumption and accelerated battery drain. Finally, a WebSocket connection is inherently stateful, which significantly complicates horizontal scaling and connection lifecycle management: restarting or redeploying servers leads to the loss of active sessions, which requires additional infrastructure for their redistribution and restoration.

Server-Sent Events (SSE) are a standardized HTML5 mechanism for unidirectional streaming of data from the server to the client over the HTTP protocol. In contrast to WebSockets, SSE use a single long-lived HTTP connection without a complex handshake and protocol switch, which simplifies integration with the existing network and security infrastructure. From the efficiency standpoint, SSE provide a favorable combination of low overhead and implementation simplicity: the use of a single long-lived channel reduces protocol overhead, while the connection itself remains sufficiently friendly to proxies and firewalls. High-concurrency benchmarks (including scenarios with dozens of parallel connections per tab) demonstrate near-linear scalability and, which is fundamentally important for the client side, lower CPU load compared to WebSockets under moderate data volumes and update frequencies [13, 16].

From the perspective of the TDSI pattern, the unidirectionality of SSE is not a limitation but rather becomes an architectural advantage. State invalidation implies rare but semantically significant signals from the server to the client, while the requests for the data themselves are executed by conventional means (for example, via fetch over HTTP/HTTPS). In this model, it is sufficient for the server to reliably and efficiently push only cache invalidation notifications toward the client, without maintaining a heavyweight full-duplex channel for continuous exchange. This reduces system complexity, simplifies traversal of corporate security boundaries, and makes SSE a natural transport layer for implementing Telemetry-Driven State Invalidation under real-world constraints on resources and network infrastructure.

Below, Table 1 presents the comparative characteristics of the protocols for invalidation tasks.

Table 1. Comparative characteristics of protocols for invalidation tasks (compiled by the author based on [18]).

Characteristic	HTTP Polling	Long-Polling	WebSockets (WS)	Server-Sent Events (SSE)
Direction	Request-Response	Request-Response (Delayed)	Full Duplex	Server-to-Client (Push)
Latency	High	Medium	Low (~15ms)	Low (~25ms)
Connection overhead	Extreme (per request)	High	Low (1 TCP)	Low (1 HTTP)
Battery consumption	High (frequent wake-ups)	High	High (Keep-alive)	Moderate
Recovery	Manual	Manual	Manual	Automatic (Native)
Throughput	Low	Low	Maximum	High
Infrastructure complexity	Low	Medium	High	Low (Standard HTTP)

The TDSI pattern fundamentally inverts the traditional allocation of responsibility for state updates. Instead of the client heuristically guessing the moment to request fresh data or the server indiscriminately broadcasting all changes to all subscribers, the system relies on telemetry feedback and performs targeted, context-dependent cache invalidation. Thus, the decision on whether an update is required is no longer hard-wired into client-side or server-side logic and becomes a consequence of the observable behavior of the system and the user.

Architecturally, TDSI implements a closed control loop in which three interrelated components can be distinguished. First, the sensing layer is represented by the Client Telemetry Agent — an SPA-integrated module based on OpenTelemetry that enriches outgoing HTTP requests with state metadata or periodically sends heartbeats (heartbeats). A typical telemetry vector may have the following form:

{ ViewID: "product_page_123", DataVersion: "v45.1", LastSync: 1715623000, UserActivity: "scroll/active" }

The transmission of these attributes is implemented via the W3C Trace Context and Baggage mechanisms, which makes it possible to propagate the data version context along the request chain without modifying existing API contracts between services.

Second, on the server side there is a controller, the Drift Detection Engine, deployed as a sidecar proxy, a middleware layer, or an autonomous stream processor (for example, based on Flink or Kafka Streams), which analyzes the continuous telemetry stream. Its core algorithm compares the DataVersion received from the client with the CanonicalVersion in the database; when a predefined threshold of acceptable divergence is exceeded, an invalidation event is generated. An important feature is the semantic differentiation of changes: the engine can distinguish critically significant modifications (for example, a change in price or product availability status) from minor cosmetic edits (such as fixing a typo in the description), thereby avoiding redundant user interface updates and network calls without real business value.

Third, the role of actuator is played by the invalidation signal delivery channel, preferably implemented on top of SSE. The payload is a lightweight notification of the form:

{ type: "INVALIDATE", resource: "inventory/123", urgency: "high" }.

Upon receiving such a message, the client does not perform a coarse page reload but instead initiates a targeted background re-fetch of the corresponding resource over standard HTTP (for example, via TanStack Query or Apollo Client), relying on the existing caching mechanisms of the browser and the CDN.47 This makes it possible to integrate TDSI into a typical frontend infrastructure with minimal changes and without breaking established data access patterns.

The key novelty of TDSI manifests itself in the abandonment of a statically defined Time-to-Live (TTL) for cached data. Any fixed TTL inevitably turns out to be an incorrect compromise: for highly dynamic resources it is too large, causing state staleness, while for stable ones it is excessively small, provoking meaningless updates and increased load. Instead, TDSI proposes a dynamic TTL algorithm parameterized by the observed data volatility and the level of user interest. Formally, the next cache lifetime interval is defined by the expression.

If the user becomes inactive and the tab goes into the background, the mechanism exponentially sparsifies checks, reducing the volume of telemetry, network traffic, and energy consumption. When the user returns (visibilitychange event), the client immediately sends an up-to-date telemetry packet, triggering drift detection and, if necessary, rapid invalidation of the stale state.

Particular attention in the context of TDSI is paid to managing optimistic updates. The Optimistic UI pattern deliberately creates a temporary divergence of versions: the client visualizes a future state version vopt that has not yet been committed on the server. In the proposed scheme, the client explicitly marks such states in telemetry, for example, via a version of the form DataVersion: v45+opt. If the server successfully applies the corresponding operation, it confirms the transition of the system to the new canonical version v46. Otherwise — when the operation is rejected by the server (validation error, business conflict) or in the event of a network failure — the Drift Detection Engine records a mismatch between the actual server state v45 and the optimistic client version vopt. In response, a forced rollback signal is generated, which returns the interface to the consistent version and thereby eliminates the hallucination of the user interface [9, 12].

Implementing TDSI in practice requires tight integration of modern observability standards directly into the logic of the frontend application and its interaction with the backend. In this context, OpenTelemetry (OTel), traditionally used for tracing server-side components, turns out to be a convenient foundation for client instrumentation as well (RUM). For the needs of TDSI, OTel semantic conventions are extended with attributes that reflect data versions and characteristics of the local state. Conceptually, the lifecycle of viewing a specific data set in an SPA is modeled as a Span — an interval during which the user interacts with a particular view, and the system accumulates and transmits the telemetry metadata associated with it.

It is critically important to minimize the overhead of the OTel agent on the client. The use of lightweight exporters and data batching makes it possible to keep the impact on the Main

Thread within acceptable bounds, without blocking the UI. For the Server -> Client communication channel, Server-Sent Events are recommended due to the native browser support for reconnections (see Fig. 1).

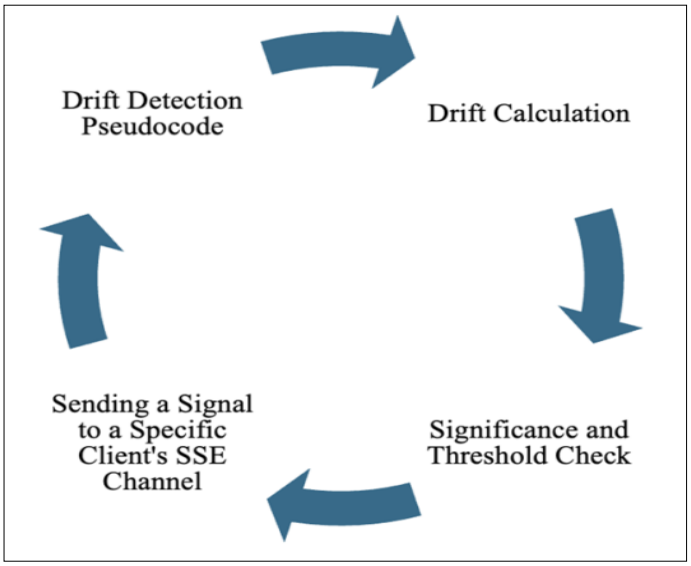

Fig. 1. Server logic (compiled by the author based on [12]).

This approach makes it possible to avoid the Thundering Herd problem (when all clients request data simultaneously), since the server can control the sending of signals by adding random delay (jitter) or prioritizing active users.

TDSI works very well with a CDN. An invalidation signal forces the client to make a standard HTTP request. This request can be served by an edge node (CDN) if fresh data are available there. If the CDN is also stale, the request goes to the origin. This creates multi-layer protection:

– Drift Detector: is aware of the freshest data.

– Client: knows that its data are stale.

– CDN: serves the update, offloading the database.

Such a scheme is used in high-load systems such as Slack (Edge Cache, Flannel) to reduce latency.

The effectiveness of patterns related to TDSI is confirmed by empirical data from the largest technological ecosystems, where similar solutions are deployed under strict requirements for latency, state coherence, and resilience at the scale of millions of concurrent users. For logistics platforms, the cost of errors in stale data is extremely high: any discrepancy between the actual state of the system and what the user sees immediately turns into incorrect ETAs, wrong order statuses, and a loss of trust in the service. Under these conditions, players such as Uber and DoorDash fundamentally cannot rely on caching with a fixed TTL as the primary mechanism of state reconciliation. Uber has built an architecture in which the mobile client continuously emits normalized telemetry (Impression, Tap, Scroll) into a separate shadow pipeline. This event stream is treated not as a by-product for subsequent offline analytics but as the primary source of truth about the user context. On the server side, this telemetry is used to form an up-to-date representation of the session state, which makes it possible to selectively push to the client changes in the status of drivers, orders, and interface components. As a result, the presentation layer is logically separated from the business logic, while both parts operate on a single server source of truth, ensuring consistent behavior and user interface across all platforms [11]. In DoorDash, the core of the real-time architecture is the Flink-based streaming engine (the Riviera system), on which feature computation is performed on the fly. Moving state computation and its evolution to the backend, as well as using a push model for delivering updates to the client, has made it possible to achieve response times of predictive models of less than 100 ms, which is a critical condition for accurate ETA calculation and prompt reaction to the dynamics of demand and courier load [10].

A similar class of problems has emerged in the domain of collaborative applications. Slack faced systemic desynchronization: the state of channels, message lists, and read indicators differed noticeably across clients, especially in scenarios with frequent reconnects, device changes, and clients waking from sleep. The response was the introduction of the DataProviders framework and a peripheral cache (Edge Cache, Flannel). The client application stopped being treated as an autonomous local store and was reinterpreted as a window onto a cache located at the network edge. The server-side pipeline took over coherence management: it pushes metadata about changes (versions, offset markers, invalidations), while the client loads the actual content in a deferred, lazy mode as it is actually needed. This architectural restructuring made it possible to eliminate reconnection storms, when many clients simultaneously attempt to restore state, and to guarantee that after a device wakes from sleep the user immediately sees the correct and complete status of messages without encountering artifacts of stale cache [12].

In e-commerce, the freshness of inventory has a direct and easily measurable financial expression: any discrepancies between actual availability and the displayed information inevitably lead to lost orders and an increase in transactional failures. According to data for 2024, the average inventory accuracy is estimated at 83%, which means that in approximately 17% of user scenarios the storefront or product page displays an incorrect availability state [8]. Against this background, the abandoned-cart rate reaches 70.19%, and a significant share of the technical causes behind this figure is the situation in which an Out of Stock error occurs at the final stages of checkout: the item was considered available at the moment it was added to the cart, but by the time of payment the actual inventory state had already changed, while the client cache continued to reflect stale data. Embedding controlled invalidation into the client architecture according to TDSI principles makes it possible to radically shorten the lifetime of a phantom product:

instead of minutes defined by a standard TTL, the state is corrected within milliseconds after the change on the server. Achieving inventory accuracy at the level of 95% and above leads to the recovery of a substantial share of previously lost revenue; the overall market estimates the total losses from abandoned carts at about 260 billion dollars annually [6].

The results of simulations and A/B experiments conducted for adaptive polling and controlled invalidation show stable quantitative gains across key metrics. Adaptive polling schemes that take into account periods of user inactivity and dynamically change the request frequency make it possible to reduce network traffic consumption by 40–60% compared with the traditional fixed polling model. At the same time, eliminating delays in updating prices and statuses (keeping the total latency of the server → client chain below one second) prevents an approximately 7% decrease in conversion that would otherwise be caused by discrepancies between the data displayed to the user and the actual state of the system. These effects show that TDSI-like patterns influence not only the technical characteristics of the system but also first-order business metrics.

At the same time, TDSI should be regarded as a high-level and costly architectural pattern in terms of engineering complexity. Its adoption requires building a full-featured telemetry pipeline, deploying stream-processing infrastructure on the backend, and maintaining a robust SSE channel for bidirectional state exchange. Economically and organizationally, this approach is justified primarily for class A data — inventory, prices, wallet balances, document state in collaborative editing, and other business-critical entities that are sensitive to latency and coherence errors. For static or quasi-static content (blog materials, textual product descriptions, marketing pages), the classical distribution scheme via a CDN with a configurable TTL remains a cheaper and sufficient solution that does not require a profound revision of client–server interaction [5, 8].

An additional constraint is imposed by regulatory privacy requirements. The collection of telemetry on user behavioral activity (scrolling, tab visibility, interface interaction patterns) inevitably falls within the scope of regimes such as GDPR and CCPA, since it may be classified as the processing of personal or quasi-identifying data. The use of RUM signals strictly for technical purposes — performance optimization, failure diagnostics, and adaptation of data delivery strategies — in most cases can be justified on the basis of Legitimate Interest. However, the data must be anonymized or at least pseudonymized and cannot be used for behavioral profiling or targeting without the explicit consent of the user. Accordingly, the TDSI architecture must be designed so that only technical identifiers, excluding the presence of PII (Personally Identifiable Information), are transmitted via telemetry channels.

Finally, significant constraints are imposed by the specifics of mobile platforms. Maintaining a persistent SSE connection, although less resource-intensive than a WebSocket session, still leads to regular radio module activity and, consequently, increased energy consumption, which directly affects the user experience. For mobile clients, TDSI should rely on an aggressive Sleep strategy: when the application goes into the background, the long-lived SSE connection should be terminated, shifting the delivery of critically important events (for example, changes in order status or balance) to the push notification infrastructure (APNS/FCM). This hybrid approach makes it possible to preserve the key advantages of controlled invalidation and high data freshness while simultaneously satisfying the strict constraints on battery and network resources of devices.

CONCLUSION

The problem of state drift in single-page applications (SPA) is not an implementation anomaly but a natural consequence of the fundamentally distributed nature of the modern web. Attempts to compensate for this drift using classical approaches — through static TTLs and rigid polling schemes — have effectively reached the asymptote of their usefulness: any gain in data freshness is achieved at the cost of an unacceptable increase in infrastructure load, while reducing traffic inevitably leads to a rise in the staleness of the displayed state.

The proposed Telemetry-Driven State Invalidation (TDSI) pattern offers not an incremental improvement of existing practices but a paradigm shift: abandoning blind periodic updates in favor of data-driven, deliberate state synchronization. In such an architecture, telemetry becomes not a by-product but a full-fledged feedback channel that allows the system to adapt the frequency and granularity of invalidation to the actual user activity and the factual dynamics of the data. In other words, the application state begins to breathe in sync with user interaction patterns and the rate of change on the backend side.

The analysis carried out shows that applying TDSI makes it possible to simultaneously solve several previously competing problems. First, guarantees of data freshness are provided, that is, a low Age of Information (AoI), comparable to the characteristics of systems operating in a mode close to real-time. Second, due to adaptive behavior, parasitic traffic is reduced and the load on server components decreases, since invalidation is initiated when it actually makes sense rather than according to a rigid schedule. Third, the frequency of desynchronization between the displayed and actual state is reduced, which directly improves the user experience and reduces financial losses associated with errors caused by stale data.

REFERENCES

1. Ollila, R., Mäkitalo, N., & Mikkonen, T. (2022). Modern web frameworks: A comparison of rendering performance.

Journal of Web Engineering, 21(3), 789-813. https://doi.org/10.13052/jwe1540-9589.21311.

2. Is Single Page Application still in demand in 2024 - DigitilizeWeb. Retrieved From: https://www.digitilizeweb.com/is-single-page-application-still-in-demand-in-2024/ (date accessed: October 17, 2025).
3. State of Frontend 2024 - The Software House. Retrieved From: https://tsh.io/state-of-frontend (date accessed: October 17, 2025).
4. van der Linde, A., Leitão, J., & Preguiça, N. (2020). Practical client-side replication: Weak consistency semantics for insecure settings. Proceedings of the VLDB Endowment, 13(12), 2590-2605. https://doi.org/10.14778/3407790.3407847.
5. Krechowicz, A., Deniziak, S., & Łukawski, G. (2021). Highly scalable distributed architecture for NoSQL datastore supporting strong consistency. IEEE Access, 9, 69027-69043.https://ieeexplore.ieee.org/abstract/document/9424000#:~:text=10.1109/ACCESS.2021.3077680.
6. Jiang, D., Zhang, G., & Wang, L. (2021). Empty the shopping cart? The effect of shopping cart item sorting on online shopping cart abandonment behavior. Journal of Theoretical and Applied Electronic Commerce Research, 16(6), 1973-1996.
7. 50 Cart Abandonment Rate Statistics 2025. Retrieved From: https://baymard.com/lists/cart-abandonment-rate (date accessed: October 18, 2025).
8. 10 Causes of Inventory Discrepancies and How to Prevent Them . Retrieved From: https://www.netsuite.com/portal/resource/articles/inventory-management/inventory-discrepancies.shtml (date accessed: October 18, 2025).
9. Chowdhury, A. R., Paul, R., & Rozony, F. Z. (2025). A systematic review of demand forecasting models for retail e-commerce enhancing accuracy in inventory and delivery planning. International Journal of Scientific Interdisciplinary Research, 6(1), 01-27. https://doi.org/10.63125/mbbfw637.
10. Tunmise, A., Esther, A., Franca, G., & Paranjabey, H. (2025). Automated Feature Engineering and Transformation in Real-Time ETL Pipelines for Predictive Modeling, 1-10.
11. How Uber Standardized Mobile Analytics for Cross-Platform Insights. Retrieved From: https://www.uber.com/blog/how-uber-standardized-mobile-analytics/ (date accessed: October 19, 2025).
12. Client Consistency at Slack: Beyond Libslack | Engineering at Slack. Retrieved From: https://slack.engineering/client-consistency-at-slack-beyond-libslack/ (date accessed: October 20, 2025).
13. Shackleton, W., Cohn-Gordon, K., Rigby, P. C., Abreu, R., Gill, J., Nagappan, N., ... & Saindon, J. (2023, November). Dead code removal at meta: Automatically deleting millions of lines of code and petabytes of deprecated data. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 1705-1715). https://doi.org/10.1145/3611643.3613871.
14. Saidani, I., Ouni, A., Ahasanuzzaman, M., Hassan, S., Mkaouer, M. W., & Hassan, A. E. (2022). Tracking bad updates in mobile apps: A search-based approach. Empirical Software Engineering, 27(4), 81.
15. Mhaisen, N., Iosifidis, G., & Leith, D. (2023). Online caching with no regret: Optimistic learning via recommendations. IEEE Transactions on Mobile Computing, 23(5), 5949-5965. https://doi.org/10.1109/TMC.2023.3317943
16. Song, Y., Gebhardt, C., Liao, Y. C., & Holz, C. (2025, September). Preference-Guided Multi-Objective UI Adaptation. In Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology (pp. 1-13). https://doi.org/10.1145/3746059.3747645.
17. Murley, P., Ma, Z., Mason, J., Bailey, M., & Kharraz, A. (2021, April). Websocket adoption and the landscape of the real-time web. In Proceedings of the Web Conference 2021 (pp. 1192-1203).
18. Van de Vyvere, B., Colpaert, P., & Verborgh, R. (2020, June). Comparing a polling and push-based approach for live open data interfaces. In International Conference on Web Engineering (pp. 87-101). Cham: Springer International Publishing.