REFINE: Rate-limited Event Fetching for Interactive Network Experiences

Technical Documentation Team
Reynard Project

September 7, 2025

Abstract

This paper introduces REFINE, a novel system designed to enhance the responsiveness and efficiency of client-server interactions in web applications by implementing modular and reliable rate-limiting for GET requests. Focusing on polling-based status updates, as exemplified by the download manager in the YipYap application, REFINE proposes a generic client-side mechanism to prevent excessive network requests, reduce server load, and improve overall user experience. We detail the current polling implementation, identify its limitations, and present a new composable-based architecture that enables adaptive polling intervals, dynamic rate adjustment, and centralized control over network resource consumption.

1 Introduction

In modern web applications, real-time feedback and status updates are crucial for a rich user experience. Polling, where the client periodically requests status from the server, is a common pattern for achieving this. However, unconstrained polling can lead to inefficient resource utilization, increased server load, and unnecessary network traffic. The YipYap application, which includes features like model downloads and indexing, currently employs fixed-interval polling for status checks. This document outlines REFINE, a system designed to introduce intelligent rate-limiting to these polling mechanisms, ensuring optimal performance and resource management without compromising responsiveness.

2 Current Polling Implementation (YipYap Download Manager Example)

The YipYap application utilizes a `ModelDownloadManager` on the backend (`app/model_download_manager.py`) to manage the state and progress of background model downloads. The frontend retrieves this status via a dedicated REST API endpoint, `/api/joycaption/download-status`, exposed by `app/main.py`.

On the client-side, the `src/contexts/app.tsx` file contains the `checkJoyCaptionDownload` function, which is responsible for initiating these status checks. This function is called periodically using `window.setInterval` with a fixed interval of 3000 milliseconds (3 seconds).

```
// src/contexts/app.tsx
// ... existing code ...
      // Start polling if not already started
      if (!downloadCheckInterval) {
        console.log("[JoyCaption] Starting polling for download status.");
        downloadCheckInterval = window.setInterval(
          checkJoyCaptionDownload,
          3000 // Check every 3 seconds for smoother progress updates
        );
      }
// ... existing code ...
```

While effective for providing real-time updates, this fixed-interval polling approach does not adapt to network conditions, server load, or the actual rate of status changes. This can result in unnecessary GET requests when the status is unlikely to change rapidly or, conversely, a delayed update if a critical status change occurs between polling intervals.

3 Limitations of Current Approach

1. **Fixed Polling Interval:** The static 3-second interval does not dynamically adjust to the download's progress or network conditions. During periods of slow progress or server-side delays, excessive requests may be made.
2. **Increased Server Load:** Frequent polling from multiple clients or for multiple concurrent operations can collectively burden the backend, leading to performance degradation.
3. **Network Inefficiency:** Unnecessary requests consume bandwidth and client resources, particularly on mobile devices or constrained networks.
4. **Lack of Centralized Control:** Each polling mechanism (e.g., for downloads, indexing) is managed independently, making it difficult to implement a unified rate-limiting strategy across

the application.

4 Proposed REFINE System Design

The REFINE system proposes a centralized, composable-based approach to rate-limit GET requests for status checks. This system will introduce an adaptive polling mechanism that can dynamically adjust the polling interval based on various factors, such as the perceived change rate of the status, network latency, and configurable thresholds.

4.1 Core Components

1. `useRateLimitedFetcher` **Composable:** This new SolidJS composable will encapsulate the logic for making rate-limited GET requests. It will accept a fetch function, an initial interval, and configuration options for adaptive rate limiting.
    - **Adaptive Interval Calculation:** The composable will dynamically adjust the polling interval. For instance, if the status has not changed for a certain number of checks, the interval could gradually increase (e.g., exponential backoff). Conversely, if the status is changing rapidly (e.g., during an active download), the interval could decrease to provide more real-time updates, up to a minimum threshold.
    - **Concurrency Control:** The composable will ensure that only one request is in flight at a time for a given endpoint, preventing race conditions and duplicate requests.
    - **Centralized Configuration:** The `useRateLimitedFetcher` will allow for global configuration of rate-limiting parameters, ensuring consistent behavior across different polling mechanisms.
2. **Backend Rate Limiting (Conceptual):** While the primary focus of REFINE is client-side rate limiting, the paper will acknowledge the importance of complementary backend rate-limiting measures (e.g., token bucket algorithm) to protect server resources from malicious or overly aggressive clients.

4.2 Implementation Details (`useRateLimitedFetcher`)

The `useRateLimitedFetcher` composable would reside in `src/composables/useRateLimitedFetcher.ts`. It would manage its own internal state for the current polling interval, the last successful fetch time, and potentially a history of status changes to inform adaptive adjustments.

Here's a conceptual outline of the `useRateLimitedFetcher` composable:

```
1  // src/composables/useRateLimitedFetcher.ts
2  import { createSignal, onCleanup, createEffect } from 'solid-js';
```

```
3
4  type FetcherFunction<T> = () => Promise<T>;
5
6  interface RateLimiterOptions {
7    initialIntervalMs?: number;
8    minIntervalMs?: number;
9    maxIntervalMs?: number;
10   // Factor by which to increase interval if no change detected
11   increaseFactor?: number;
12   // Factor by which to decrease interval if change detected
13   decreaseFactor?: number;
14   // How many stable checks before increasing interval
15   stableChecksThreshold?: number;
16 }
17
18 export function useRateLimitedFetcher<T>(
19   fetcher: FetcherFunction<T>,
20   options?: RateLimiterOptions
21 ) {
22   const effectiveOptions = {
23     initialIntervalMs: 3000,
24     minIntervalMs: 1000,
25     maxIntervalMs: 30000,
26     increaseFactor: 1.5,
27     decreaseFactor: 0.8,
28     stableChecksThreshold: 3,
29     ...options,
30   };
31
32   const [data, setData] = createSignal<T | null>(null);
33   const [isLoading, setIsLoading] = createSignal(false);
34   const [error, setError] = createSignal<any | null>(null);
35   const [currentInterval, setCurrentInterval] =
     ↪ createSignal(effectiveOptions.initialIntervalMs);
36   const [stableChecks, setStableChecks] = createSignal(0);
37
38   let intervalId: number | undefined;
39   let lastFetchedData: T | null = null;
40
41   const fetchData = async () => {
42     if (isLoading()) return; // Prevent concurrent fetches
43
44     setIsLoading(true);
45     setError(null);
46
```

```
    try {
      const result = await fetcher();
      setData(() => result); // Use a function to set signal for direct access to value

      // Adaptive interval logic
      if (JSON.stringify(result) === JSON.stringify(lastFetchedData)) {
        setStableChecks(prev => prev + 1);
        if (stableChecks() >= effectiveOptions.stableChecksThreshold) {
          setCurrentInterval(prev => Math.min(prev * effectiveOptions.increaseFactor,
          ↪ effectiveOptions.maxIntervalMs));
        }
      } else {
        setStableChecks(0);
        setCurrentInterval(prev => Math.max(prev * effectiveOptions.decreaseFactor,
        ↪ effectiveOptions.minIntervalMs));
      }
      lastFetchedData = result;

    } catch (err) {
      setError(err);
      // On error, perhaps reset interval to initial or increase more aggressively
      setCurrentInterval(effectiveOptions.initialIntervalMs);
    } finally {
      setIsLoading(false);
    }
  };

  const startPolling = () => {
    stopPolling(); // Ensure no multiple intervals
    intervalId = window.setInterval(fetchData, currentInterval());
  };

  const stopPolling = () => {
    if (intervalId) {
      window.clearInterval(intervalId);
      intervalId = undefined;
    }
  };

  // Effect to manage polling based on currentInterval
  createEffect(() => {
    stopPolling();
    startPolling();
  });

```

```
90    onCleanup(() => {
91      stopPolling();
92    });
93
94    return { data, isLoading, error, refetch: fetchData, stopPolling, startPolling };
95  }
```

By integrating this `useRateLimitedFetcher` into components requiring status updates, such as the JoyCaption download status, we can achieve:

- **Reduced Network Overhead:** Fewer unnecessary requests during periods of inactivity.
- **Improved Responsiveness:** Faster updates when status changes are frequent.
- **Centralized Control:** A single, reusable composable for managing polling logic across the application.
- **Modularity:** Easy to apply to any polling-based status check without modifying core application logic.

5 Applications of REFINE in YipYap

Beyond the `ModelDownloadManager` example, several other components within the YipYap application can significantly benefit from integrating the REFINE system. By replacing fixed-interval polling with the adaptive `useRateLimitedFetcher` composable, we can further optimize network resource consumption and improve the overall responsiveness of these features.

5.1 Performance Dashboard Metrics

The `src/components/Debug/PerformanceDashboard.tsx` component periodically fetches various performance metrics (e.g., memory usage, browser responsiveness, frame rate) to display a real-time overview of the applicationś performance. Currently, these metrics are updated at a fixed interval using `window.setInterval`.

```
1  // src/components/Debug/PerformanceDashboard.tsx
2  // ... existing code ...
3          updateInterval = window.setInterval(async () => {
4              // Update current metrics if monitoring
5              if (performanceMonitor.isMonitoring()) {
6                  setCurrentMetrics(performanceMonitor.metrics());
7                  setWarnings(performanceMonitor.warnings());
8              }
9  // ... existing code ...
```

Applying REFINE here would allow the polling interval to adapt based on whether the dashboard is actively viewed or if the metrics are changing rapidly. For instance, if the application is idle and performance metrics are stable, the polling interval could increase, reducing unnecessary computations and network activity. Conversely, if the system is under load and metrics are fluctuating, the interval could decrease to provide more granular feedback.

5.2 Scroll Performance Monitor Memory Usage

The `src/composables/useScrollPerformanceMonitor.ts` composable includes a mechanism to periodically measure memory usage as part of its performance monitoring capabilities. This is currently implemented with a fixed 1-second interval.

```
// src/composables/useScrollPerformanceMonitor.ts
// ... existing code ...
    if (isMonitoring()) {
      const interval = setInterval(async () => {
        const currentMemory = await performanceMonitor.measureMemoryUsage();
        setMemoryUsage(currentMemory);\
      }, 1000); // Update every 1 second
// ... existing code ...
```

Integrating `useRateLimitedFetcher` would enable adaptive polling for memory usage. When memory consumption is stable, the interval could lengthen, reducing the overhead of frequent measurements. If memory usage fluctuates significantly, the interval could shorten to capture more dynamic changes, providing more relevant data to the performance monitor without constant polling.

5.3 Indexing Status

The application features a background indexing process, and its status is regularly polled via the `src/composables/useIndexing.ts` composable. This polling mechanism is crucial for providing users with real-time updates on the progress of image indexing, which can be a long-running operation. The current implementation uses a fixed 2-second interval.

```
// src/composables/useIndexing.ts
// ... existing code ...
    statusInterval = window.setInterval(() => {
      fetchStatus();
    }, 2000); // Poll every 2 seconds
// ... existing code ...
```

REFINE would greatly enhance the indexing status polling. When indexing is active and progress is rapidly changing, the interval could be reduced to provide immediate feedback. However, if indexing

is paused, idle, or making very slow progress, the interval could be significantly increased, drastically reducing server load and client-side processing without compromising the user experience.

5.4 YapCoin Balance

The userś YapCoin balance is periodically refreshed via the `src/composables/useYapCoins.ts` composable, currently at a fixed interval of 5 minutes.

```
// src/composables/useYapCoins.ts
// ... existing code ...
    refreshTimer = setInterval(refreshBalance, REFRESH_INTERVAL);
// ... existing code ...
```

While a 5-minute interval is already relatively long, applying REFINE could further optimize this. For instance, if the user is actively making transactions, the interval could temporarily shorten to provide more up-to-date balance information. Conversely, if the user is inactive or not on a page that displays the balance, the polling could be paused or the interval significantly extended, saving network resources and battery life, particularly on mobile devices.

6 Conclusion

The REFINE system offers a pragmatic solution to optimize client-server communication in dynamic web applications. By introducing intelligent, adaptive rate-limiting to GET requests, particularly for status polling, REFINE significantly enhances application performance, reduces server strain, and provides a more fluid and efficient user experience. This modular approach, encapsulated within a reusable SolidJS composable, lays the groundwork for more resilient and scalable frontend architectures. The extension of REFINEś application to include performance metrics, indexing status, and YapCoin balance polling demonstrates its versatility and potential for widespread positive impact across the YipYap application.