Overcoming Variable Payloads to Optimize for Performance

Armin Ronacher
Principal Architect at Sentry
Armin Ronacher
Principal Architect at Sentry

- Creator of Flask, Werkzeug, Jinja and many Open Source libs
- Keep things running at Sentry, make event processing go vroom
- Got to learn to love event processing pipelines
- Juggling three lovely kids
Why Are We Here?
Sentry Generates, Processes and Shows Events

---

**Issues**

All Unresolved | 252  
---|---

- **Error** fn(src/screens/EndToEndTestsScreen)
  - Unhandled Promise Rejection
  - SENTRY-REACT-NATIVE-97 (7min ago | 7mo old)

- **Error** onPress(src/screens/EndToEndTestsScreen)
  - captureException test
  - SENTRY-REACT-NATIVE-96 (18min ago | 7mo old)

- **React Native Test Message**
  - SENTRY-REACT-NATIVE-8E (18min ago | 7mo old)

- **Error** apply[native]
  - Thrown Error
  - SENTRY-REACT-NATIVE-44 (23min ago | 16mo old)

- **Error** onPress[index]
  - Thrown Error
  - SENTRY-REACT-NATIVE-8Y (39min ago | 12mo old)

- **Error** anonymous(src/screens/EndToEndTestsScreen)
  - Unhandled Promise Rejection
  - SENTRY-REACT-NATIVE-93 (31min ago | 7mo old)

- **Error** onPress(src/screens/EndToEndTestsScreen)

---

**GRAPH:**

**24h** | **14d** | **EVENTS** | **USERS** | **ASSIGNEE**
---|---|---|---|---
... | ... | 65 | 65 |  
... | ... | 65 | 65 |  
... | ... | 98 | 98 |  
... | ... | 3 | 1 |  
... | ... | 1 | 1 |  
... | ... | 37 | 37 |  

---

P99 CONF
Sentry Generates, Processes and Shows Events

System.NullReferenceException

Object reference not set to an instance of an object.

14. public void AssertFalse() => Assert.AreEqual(true, false);
15.  [MethodImpl(MethodImplOptions.NoInlining)]
16. public void ThreadNull() => throw null;
17.  
18. Debug.Log("Throwing an instance of CustomException");
Sentry Events

- Session Updates
- Transaction Events
- Metrics
- Reports
  - Messages
  - Structured Processed Crash Reports
  - Structured Unprocessed Crash Reports
  - Minidumps
  - Third Party Crash Formats
  - User Feedback
  - Profiles
  - Attachments
  - Client Reports
Challenges

- Users want crash reports with low latency
- Variance of processing times of events from 1ms to 30 minutes
- How long an event takes, is not always known ahead of time
- What happens at the end of the pipeline can affect the beginning of it
- Part of the pipeline is an Onion that can extend closer and closer to the user
Conservative Changes
Touching Running Systems

- Sentry processes complex events from many sources
- Any change (even bugfix) can break someone’s workflow
- We are treating very carefully

Things we try to avoid doing:

- Bumping Dependencies without reason
- Rewriting services as busywork

That doesn’t mean we don’t change the pipeline, but we are rather conservative.
Terms and Things
“The Monolith”

- Written in Python
- A massive and grown Django app
- Uses celery and rabbitmq historically for all queue needs
- Still plays a significant role in the processing logic
- Uses CFFI to invoke some Rust code
Relay

- Written in Rust
- Our ingestion component
- Layers like an onion
- Stateful
- First level quota enforcement
- Aggregation
- Data normalization
- PII stripping
Symbolicator

- Written in Rust
- Handles Symbolication
  - PDB
  - PE/COFF
  - DWARF
  - MachO
  - ELF
  - WASM
  - IL2CPP
- Fetches and Manages Debug Information Files (DIFs)
  - External Symbol Servers
  - Internal Sources
Ingest Consumer

- Shovels Pieces from the Relay supplied Kafka stream onwards
  - Events
  - User Reports
  - Attachment Chunks
  - Attachments
- Does an initial routing of events to the rest of pipeline
What’s Flowing?
Ingestion Side

SDK

Relay

Event / Other

Project Config

Envelopes

Sentry

Rate Limits

(relays can be and are stacked)
Ingestion Traffic

- POP Relays accepts around 100k events/sec at regular day peak and rejects around 40k/sec
- Processing relays process around 150k events/sec at regular day peak
- Global Ingestion-Level Load Balancers see around 200k req/sec at regular peak
Processing Side

“Processing” Relay -> Kafka -> RabbitMQ -> Kafka

Bigtable -> Kafka

Clickhouse -> Postgres
Kafka Traffic

- All relay traffic makes it to different Kafka topics
- Important ones by volume:
  - Sessions/Metrics
  - Transactions
  - Error events
  - Attachments
- Based on these event types, initial routing happens
- The biggest challenge are error events
Error Event Routing

- Ahead of time, little information is available to determine how long an event will take.
- Cache status can greatly affect how long it takes:
  - JavaScript event without source maps can take <1ms
  - JavaScript event that requires fetching of source maps can take 60sec or more
  - Native events might pull in gigabytes of debug data, that’s not yet hot
- A lot of that processing still happens in legacy monolith
The Issue with Variance
Head of Line Blocking within Partition

- Fast Event
- Fast Event
- Fast Event
- Slow Event
Our Queues: Kafka and RabbitMQ

- Kafka has inherent head-of-line blocking
- Our Python consumers have language limited support for concurrency
- Writing a custom broker on top of Kafka carries risks
- Historically our answer was to dispatch from Kafka to Rabbit for high variance tasks
We’re Not Happy with RabbitMQ

- As our scale increases, we likely will move to Kafka entirely
- This switch will require us to build a custom broker
- So far the benefits of that have not yet emerged
- It works good enough for now™
Tasks on RabbitMQ

- Tasks travel on RabbitMQ queues
- Event payloads live in redis
- Python workers pick up tasks as they have capacity available
- Problem: polling workers
Polling Workers

- Some tasks poll the internal symbolicator service
- For that a Python worker dispatches a task via HTTP to the stateful symbolicator service
- Python worker polls that service until result is ready which can be minutes
- Requires symbolicators to be somewhat evenly configured and loaded
Incident: Symbolicator Tilt

- Fundamental flaw: tasks are pushed evenly to symbolicators
- Not all symbolicators respond the same
- A freshly scaled up symbolicator has cold caches
- This caused scaling up to have a negative effect on processing times
- Workaround: **cache sharing**
- Long term plan: symbolicator picks up directly from RabbitMQ or Kafka

![Diagram showing processing times for hot and cold symbolicators](image)
Backpressure Control
Implicit Backpressure Control

- Our processing queue has insufficient backpressure control
- At the head of the queue we permit almost unbounded event accumulation
- Pausing certain parts of the pipeline can cause it to spill too fast into RabbitMQ (goes to swap)
Deep Load Shedding
Pipeline Kill-Switches

- Problem: for some reason bad event data makes it into the pipeline
- Due to volume we cannot track where the data is in the pipe and we likely can’t reliably prevent it from propagating further
- Solution: flexible kill-switches
- Drop events that match a filter wherever that filter is applied
sentry killswitches pull \\nstore.load-shed-group-creation-projects \\
new-rules.txt

Before: <disabled entirely>
After:
   DROP DATA WHERE
      (project_id = 1) OR
      (project_id = 2) OR
      (project_id = 3)

Should the changes be applied? [y/N]: y
Look into Relay
Communication Channels

- Relay to Relay: HTTP
- Relay to Processing Pipeline: Kafka
- Relay state updates:
  - Relay -> Relay via HTTP
  - Relay to Internal HTTP and direct redis cache reads
Project Config Caches

- Innermost relays fetch config directly from Sentry
- Sentry itself persists latest config into redis
- Relay will always try to read from that shared cache before asking Sentry
Proactive Cache Writing

- We used to expire configs in cache liberally
- Now most situations will instead proactively rewrite configs to cache