Performance Debugging in In-Memory Data Store (A Microservice Context)

Progress Report Meeting

Hervé KABAMBA

PhD Candidate

Supervisor: Michel Dagenais

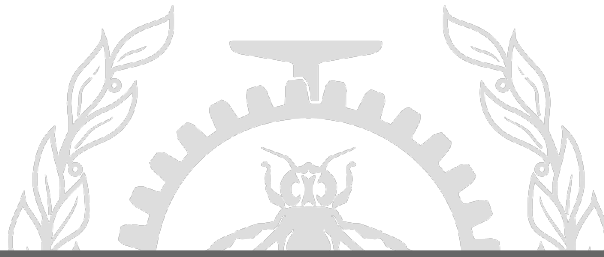
December 07, 2023

Polytechnique Montréal

Département de Génie Informatique et Génie Logiciel

Agenda

1. Introduction
2. Our Approach
3. Some results
4. Conclusion



Introduction

In-Memory Data Store

- Bring storage dimension in microservices architectures
- High availability of data
- Low latency access to data
- Databases, Messaging and caching services

Introduction(2)

Redis (Remote Dictionary Server)

An in-memory data structure used as:

- Cache

- Pub/sub

- Database

Introduction(3)

CONTEXT

- In microservice architecture in-memory data stores are generally part of the infrastructure

- They enable fast access to data

- Persistence services can be implemented

Introduction (4)

Problem:

As part of of the infrastructure, performance issues can be localized inside Redis

Distributed tracing approaches mostly target telemetry data collection from implemented microservices

In such a context, the in-memory data store appears as a black-box

Precise performance debugging approaches need to address the in-memory data store dimension

Our Approach

Building a high level model

- Redis, as an In-memory data system, has a built-in asynchronous mechanism, enabling the handling of massive amount of events

- We propose a high level model that can be generalized to other such systems, for debugging performance issues

- The model is based on the identification of key performance influencers, to capture the semantic of the system operation

- Populating the model enables the understanding of system operations, and the visualization of Redis performance impacting actors.

Our approach(2)

Model description

-The model is built on 3 levels

- The topmost level represents the Redis identified key performance influencers

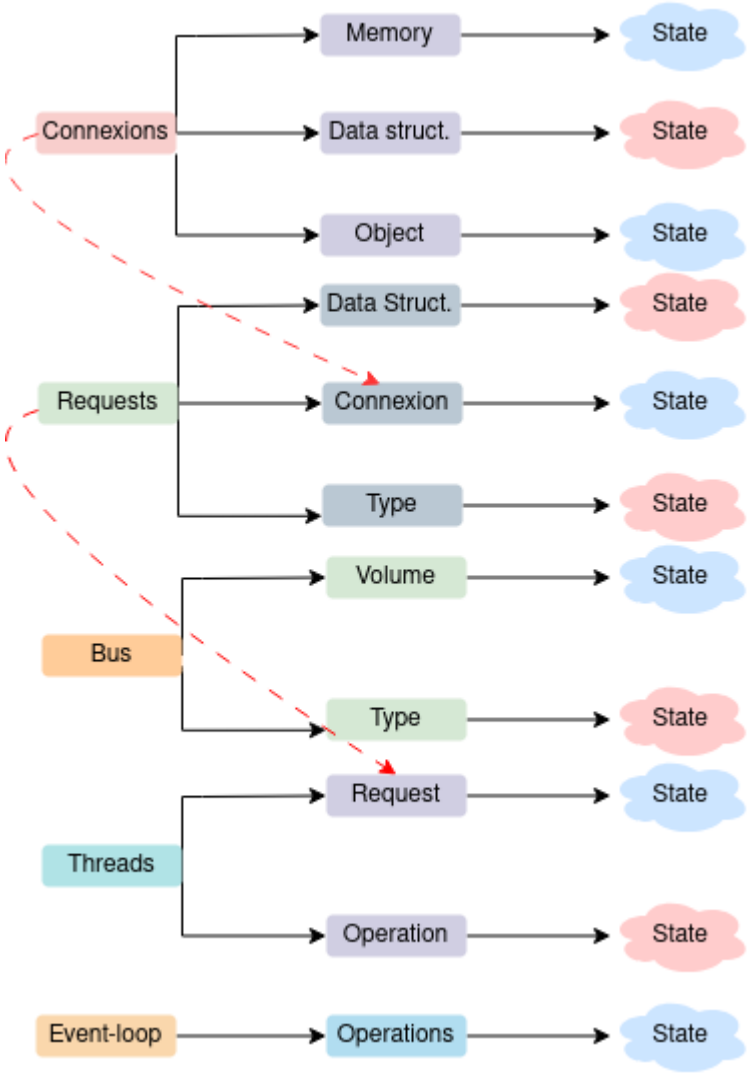
- The second level represents the resources attached to the topmost actors

- The lower level captures the statuses of each resource

The model is constructed based on the SHT (State History Tree), a highly efficient data structure featured by Trace Compass

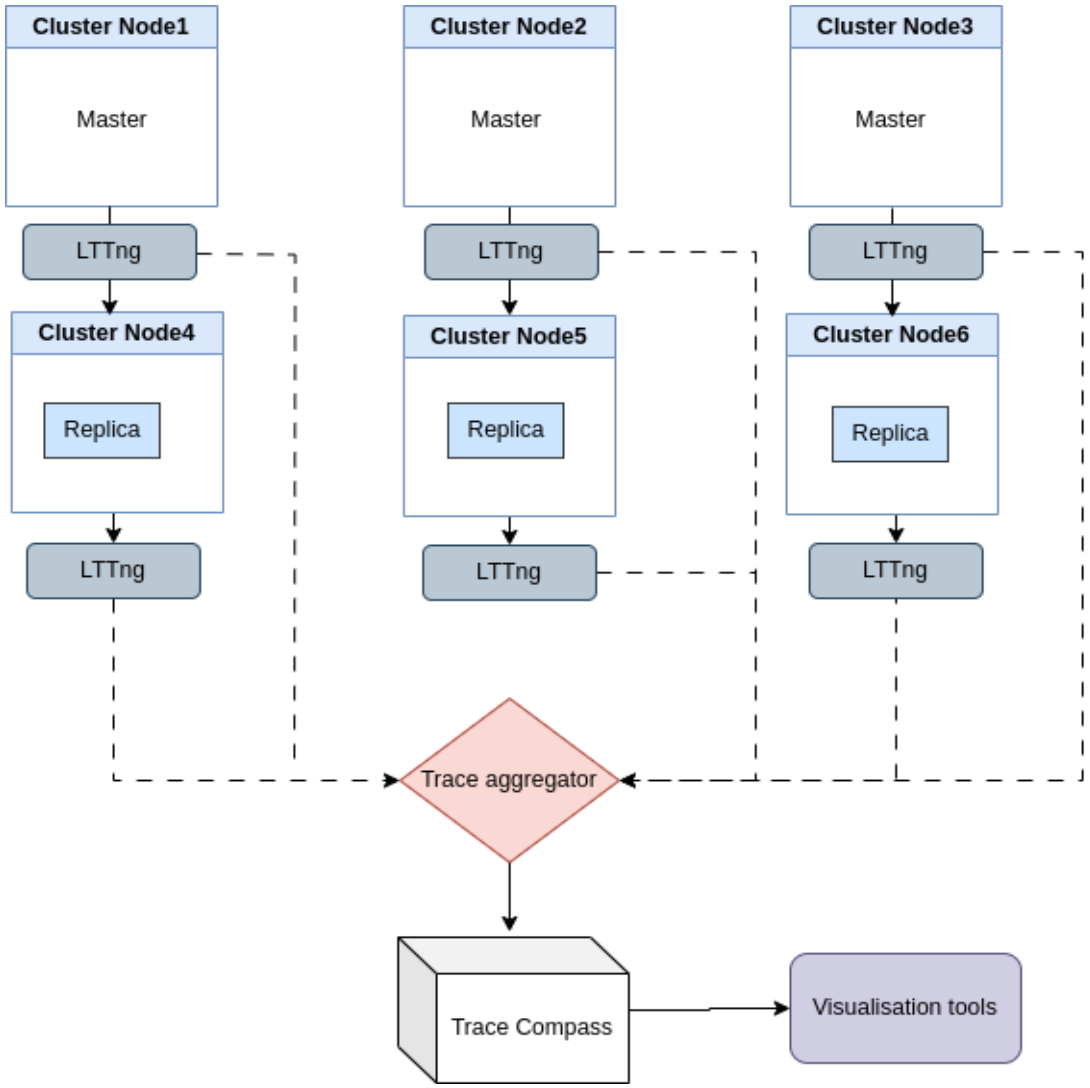
Our Approach(3)

High Level Model



Our Approach(4)

Framework architecture



Results

- In a Redis cluster, a publisher can send messages to subscribers

- Before the subscribers consume the message, the latter may traverse some cluster nodes.

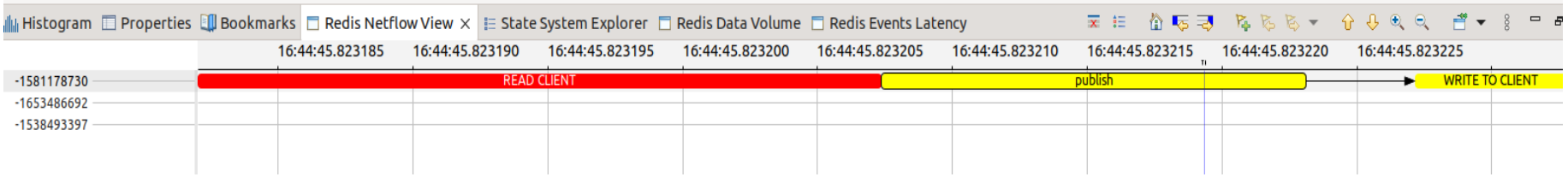
- Performance issues may happen on the cluster during the sending of the message.

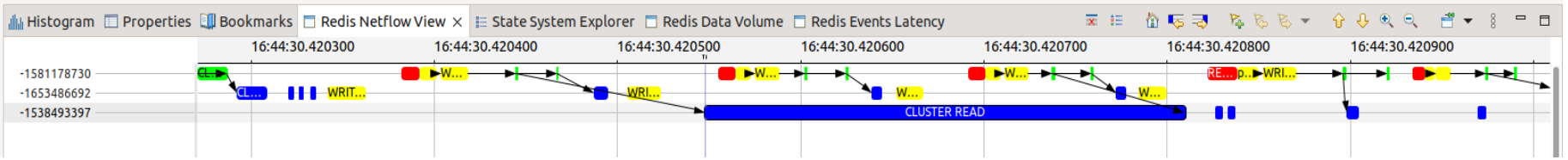
- Debugging such problems need efficient analyses and understanding the global system functioning

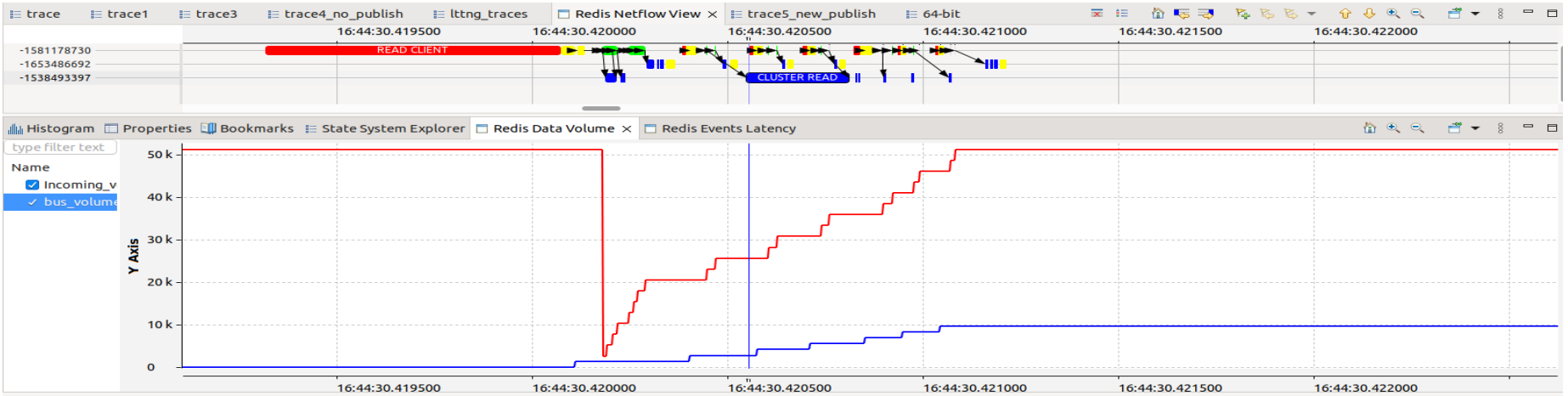
- We can leverage our model to pinpoint such performance issues when they appear

Results(2)

redis1/ust/uid/0/64-bit(6)	16:44:45.823 218 251	channel0_4	4	redis:insert_clientwrite_queue	fd=16, id=0x4, context.packet_seq_num=31, context.cpu_id=4, context._procname=redis-server, context._
redis1/ust/uid/0/64-bit(6)	16:44:45.823 219 367	channel0_4	4	redis:call_command_end	fd=16, name=publish, context.packet_seq_num=31, context.cpu_id=4, context._procname=redis-server, c
redis1/ust/uid/0/64-bit(6)	16:44:45.823 220 938	channel0_4	4	redis:end_process_command	fd=16, id=0x4, context.packet_seq_num=31, context.cpu_id=4, context._procname=redis-server, context._
redis1/ust/uid/0/64-bit(6)	16:44:45.823 223 118	channel0_4	4	redis:end_read_client_query	fd=16, nread=349, id=0x4, context.packet_seq_num=31, context.cpu_id=4, context._procname=redis-serv
redis1/ust/uid/0/64-bit(6)	16:44:45.823 224 147	channel0_4	4	redis:before_sleep_start	context.packet_seq_num=31, context.cpu_id=4, context._procname=redis-server, context._pthread_id=13
redis1/ust/uid/0/64-bit(6)	16:44:45.823 224 974	channel0_4	4	redis:handleClientsWithPendingReadsUsingThreads_start	context.packet_seq_num=31, context.cpu_id=4, context._procname=redis-server, context._pthread_id=13







Results(3)

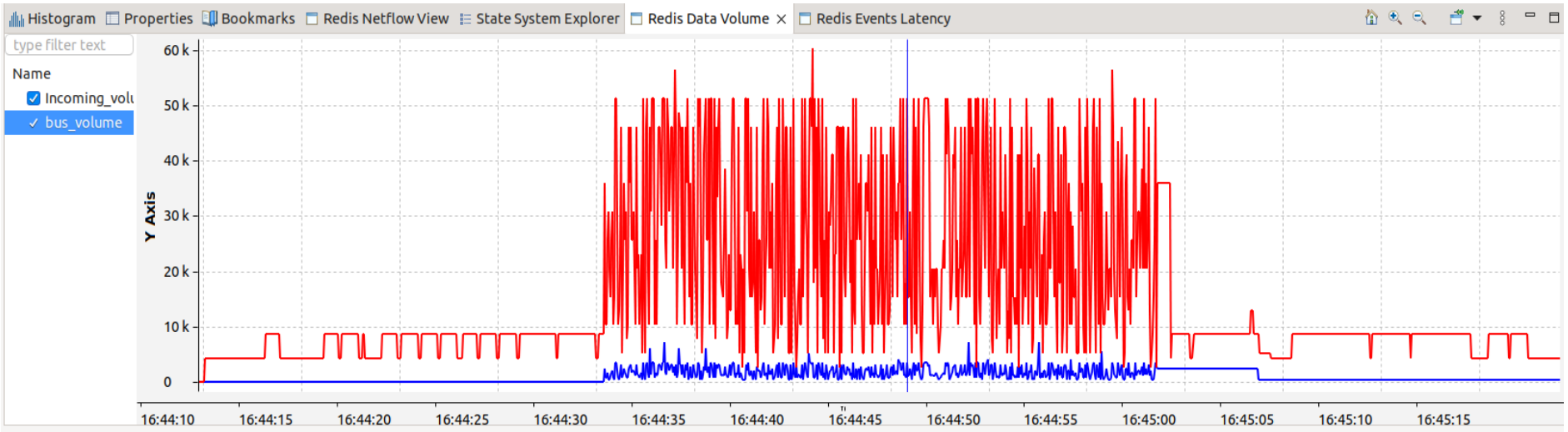
Use case 2

- Unresolved bug described on GitHub, pertains to the context of ”publish” requests sent to a Redis infrastructure consisting of a cluster.

- The amount of data sent through the cluster is 10 times bigger than the original data volume

- Our experiments reveal a bug within the GOSIP protocol used by Redis for node communication.

- First, the data is encapsulated into a structure containing a huge payload; second, it is broadcasted to all the nodes connected to the cluster. These operations generate a significant overhead on the system.



Results(3)

Use case 3 (SSL data reading bug)

Appears when Redis is compiled with TLS support and thread I/O reading activated.

- When a big item is sent and arrives in Redis in small parts, Redis only reads the needed amount of data.

- The connection object will be stored in a list for SSL I/O reading

- During the reading cycles, The SSL handler will read all data in the buffer until no data remains.

- In this case, if an event such as closing the connection happens, the latter is freed and becomes reusable, and a new reading event is triggered.

- Since the reading I/O thread was activated, the same connection is added again to the list of SSL I/O readings.

- Then, the SSL handler will be called to check if there is data remaining in the buffer.

- Since no data remains, the thread will double-free the connection and lead to a crash of Redis.

Results(4)

Step by step bug visualisation

Conclusion

- The obtained results were made possible by leveraging our presented model

- The model abstracts the asynchronous mechanism of Redis and helps comprehending its functioning

- The model can be extended and applied to other such systems (RabitMQ, ZeroMQ, MemCached, etc.)

Thank you