Technical Note

Implementation and Evaluation of a Low-Footprint and Low-Dependency Inter-Process Communication Library for Microservices

Daisuke Sugisawa *

Posted Date: 11 November 2025

doi: 10.20944/preprints202511.0596.v1

Keywords: Inter-Process Communication (IPC); MySQL; microservices

 doi:10.20944/preprints202511.0596.v1

Article

Implementation and Evaluation of a Low-Footprint and Low-Dependency Inter-Process Communication Library for Microservices

Daisuke Sugisawa

Independent Researcher, Japan; daisuke.sugisawa.xander@gmail.com

Abstract

In the modern microservice environment, the dependencies of libraries used for inter-system communication have become bloated, and conflicts and complications during build and operation have become problems. In particular, in the conventional communication architecture that depends on the MySQL database, the multi-layer dependencies included in `libmysqlclient` restrict the flexibility of system design. In this study, a replication-protocol-compatible patch was applied to the lightweight MySQL client library Trilogy, and a loosely coupled, low-footprint IPC library connecting the control plane and the data plane was implemented. The proposed method eliminates dependencies on the internal static library group of MySQL Server, while enabling binary log events to be processed directly at the application layer. Stable operation has been achieved for more than one year in a commercial system environment, and its effectiveness has been evaluated.

Keywords: Inter-Process Communication (IPC); MySQL; microservices

1. Introduction

In recent years, distributed and microservice systems have been strongly dependent on multilayered software stacks and external libraries. Therefore, with the increase in dependencies, the decrease in maintainability and the risk of conflicts have become apparent. In particular, in the MySQL [1] environment, `libmysqlclient` is highly functional, but its implementation dependency is deep, and when dealing with the replication protocol, linking with many static libraries inside the server source is required (Table 1). This structure makes it difficult to apply lightweight or embedded environments.

In contrast, the lightweight MySQL client library **Trilogy** [2] published by GitHub adopts a design optimized for asynchronous I/O while minimizing external dependencies. In this study, by extending this Trilogy with the client implementation of the replication protocol (COM_BINLOG_DUMP=0x12)[1], a low-overhead inter-process communication (IPC) module between the MySQL server and applications was realized.

2. Background and Issues

2.1. Overview of the Replication Mechanism

MySQL Server replication is mainly performed through binlog (binary log). All transaction events are recorded and transferred through the Log_event class group inside the server. The normal client library (`libmysqlclient`) implements only SQL-level communication such as COM_QUERY and COM_PING, and the replication protocol (COM_BINLOG_DUMP) is implemented only inside the server.

[1] https://github.com/trilogy-libraries/trilogy/pull/247

 doi:10.20944/preprints202511.0596.v1

2.2. Dependent Source Files and Build Issues

In order to use the replication mechanism of MySQL Server directly on the application side, it is necessary to depend on the internal source files of the main server. The list is shown below.

- **sql/log_event.cc** Binlog event generation and base class definition
- **sql/rpl_utility.cc** Common utilities for replication
- **sql/rpl_gtid_tsid_map.cc** GTID/TSID management
- **sql/rpl_gtid_misc.cc** GTID auxiliary functions
- **sql/rpl_gtid_set.cc** GTID set operation
- **sql/rpl_gtid_specification.cc** GTID definition structure
- **sql/rpl_tblmap.cc** Table Map event processing
- **sql/basic_istream.cc** Basic implementation of stream I/O
- **sql/binlog_istream.cc** Binlog input stream processing
- **sql/binlog_reader.cc** Binlog reader class
- **sql/stream_cipher.cc** Binlog encryption processing
- **sql/rpl_log_encryption.cc** Replication log encryption
- **libs/mysql/binlog/event/trx_boundary_parser.cpp** Transaction boundary analysis

These source files are designed on the premise that they are used only inside `mysqld`, and it is also necessary to depend on the static libraries in Table 1 at build time.

For this reason, even if only `libmysqlclient` is linked, compilation and linking will not pass, and it is necessary to reproduce the entire build environment of MySQL Server. In addition, since these libraries are provided under the GPL-2 license, there are also licensing constraints for use in proprietary products by static linking.

2.3. Summary of Problems

The above structural issues are summarized as follows.

- **Build Complexity:** The number of dependent objects is large, and the reproducibility of the build environment is low.
- **Maintenance Difficulty:** The internal ABI tends to change with MySQL version upgrades, making relinking difficult.
- **License Risk:** It is necessary to link GPL code, which is not suitable for MIT/BSD environments.

2.4. Issues of `libmysqlclient`

To process replication events of MySQL Server, it is necessary to link internal library groups (libmysys.a, libmysql_binlog_event.a, libmysql_serialization.a, etc.) (see Table 1). These are designed for internal use of the server, and since they are not intended to be used externally, the build configuration becomes complicated, and ABI compatibility is not guaranteed.

Table 1. mysql libraries

	Library	Function Summary
1	libmysqlclient.a	Basic C API
2	libmysys.a	Internal utility group
3	libmysql_serialization.a	Protocol serialization
4	libmysql_binlog_event.a	Binlog event construction and analysis
5	libclientlib.a	Client I/O layer
6	libmysql_gtid.a	GTID tracking
7	libjson_binlog_static.a	Binlog JSON parser

Using these involves operational risks in terms of both licensing and technology.

 doi:10.20944/preprints202511.0596.v1

2.5. MySQL Dependency and System Design Constraints

`libmysqlclient` is distributed under GPL-2, and when statically linked in commercial systems, license risks arise. Also, since the group of dependent libraries is complex, conflicts and compatibility problems are likely to occur due to differences in build environments. These are fatal in resource-constrained environments such as IoT and edge devices.

3. Proposed Method

3.1. Low-Dependency IPC Design Using Trilogy

In this study, we use the MIT-licensed lightweight MySQL client Trilogy as a foundation, and by extending it with replication protocol functionality, we construct a low-dependency and loosely coupled IPC layer (Figure 1).

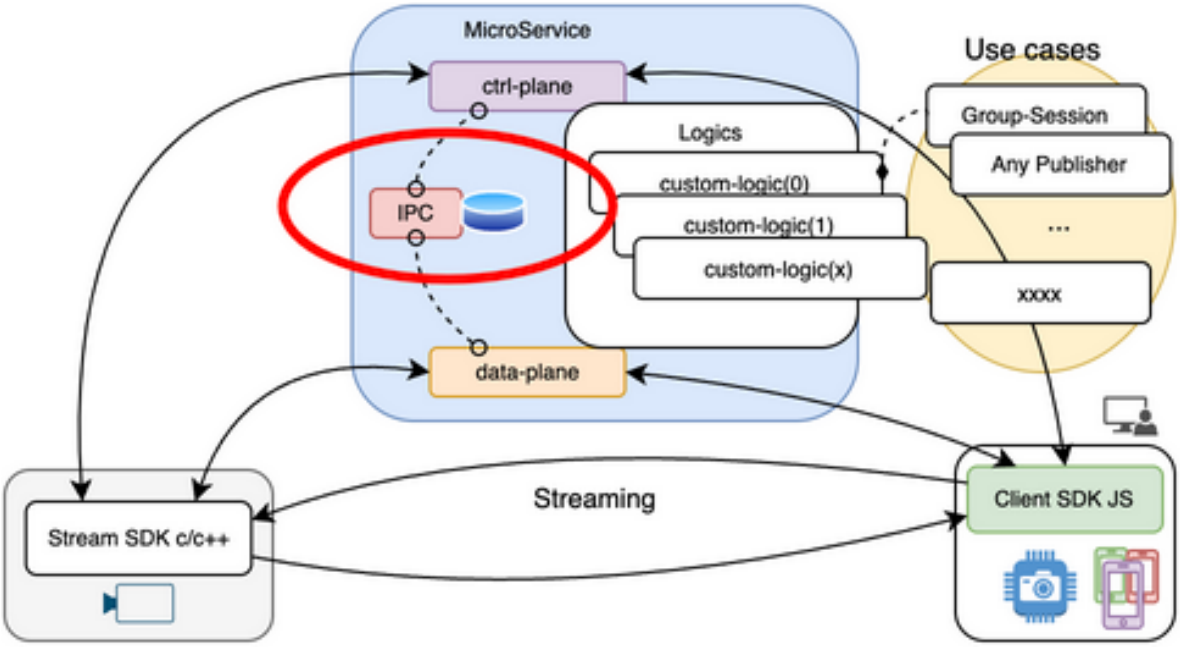

Figure 1. IPC session

`trilogy_binlog_dump()` requests a binary log stream from the MySQL server.

`trilogy_binlog_dump_recv()` sequentially receives and analyzes event packets.

The event header is analyzed by `trilogy_parse_binlog_event_packet()`, and `FORMAT_DESCRIPTION_EVENT`, `TABLE_MAP_EVENT`, and `WRITE_ROWS_EVENT` are reconstructed in user space.

By this method, it became possible to handle the replication protocol directly from the application layer without depending on the internal structure of MySQL Server.

3.2. Microservice Configuration

This IPC library functions as an intermediate layer between the control plane (ctrl-plane) and the data plane (data-plane) in a microservice architecture (Figure 2). Applications subscribe to the binary logs of the MySQL server through Trilogy and can asynchronously transmit session information and commands.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 November 2025 doi:10.20944/preprints202511.0596.v1

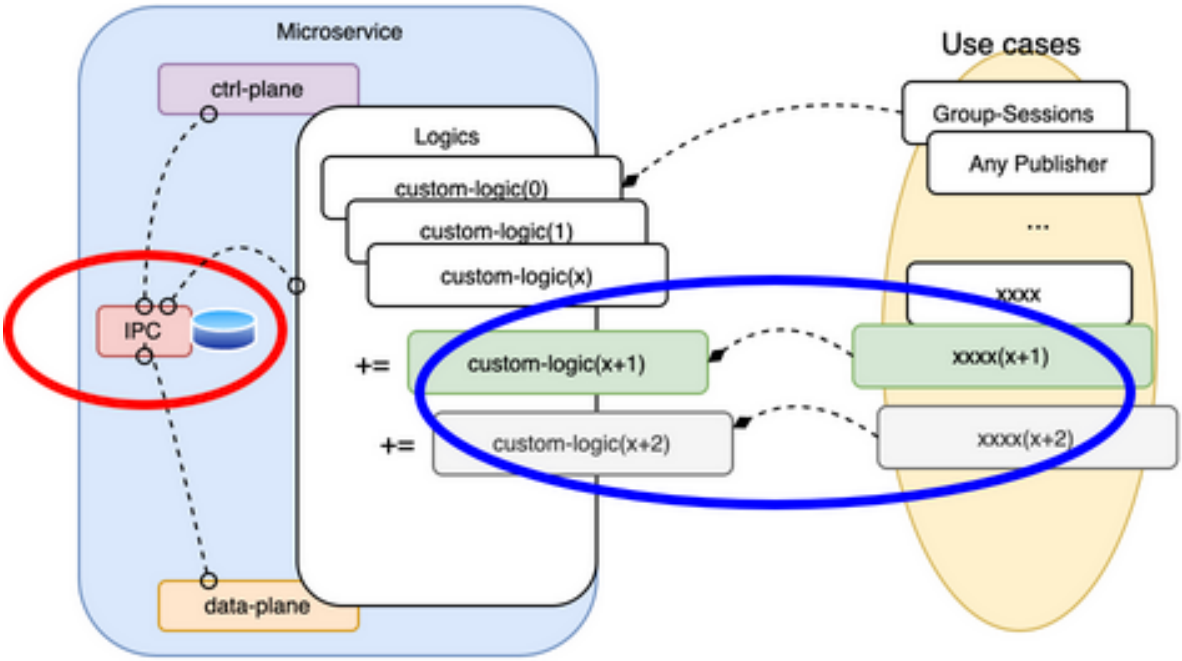

Figure 2. IPC pub-sub

3.3. Comparison of Evaluation Results

In this section, we compared the build cost and runtime overhead between the proposed method (Trilogy extension) and the conventional configuration using `libmysqlclient`. Table 1 shows the container size and build time measured in the Docker environment, and the performance comparison in a simple SQL insertion load test (noop 4ch).

Table 2. mysql Trilogy Dependencies

Item	**Conventional (libmysql-client)**	**Proposed (Trilogy extension)**
Number of dependent sources	13 or more	0
Static link libraries	7	1 (libtrilogy.a)
License	GPL-2	MIT
Build time	about 30 minutes	a few seconds
Execution performance	High speed	High speed

 doi:10.20944/preprints202511.0596.v1

Table 3. Comparison between Trilogy and libmysqlclient (Docker environment)

Item	Trilogy	mysqlclient	desc
Container Size $docker images	531 MB	2.85 GB	In order to link the replication protocol parser, it is necessary to compile and link the entire `mysql-server` source. Therefore, the container size becomes large, and in the Trilogy configuration, approximately **82% weight reduction** was achieved.
Container build cost $docker build	107.1 s	252.3 s	Because it is necessary to clone the `mysql-server` source and pre-build the replication target library group, the build cost is large. On the other hand, the Trilogy configuration reduced the build time by **about 58%**.
noop (4ch) SFU Application	6223 (333)	6169 (333)	Under the simple SQL traffic condition of `INSERT ONLY`, **approximately 1% load reduction** was confirmed. This result shows the lightweight effect of performing replication processing at the application layer.

In the Trilogy-based configuration, significant improvements were confirmed in build cost, dependent libraries, and container size compared to the `libmysqlclient` configuration.

3.4. Context Switch Comparison

In this section, we verified the differences in I/O context management methods between the proposed method (Trilogy extension) and the conventional method (`libmysqlclient`), and analyzed the effect on the number of context switches. Measurements used `pidstat -wt 1` and `strace -fc -p (pid)`, and in addition, the call path of `poll(2)` was traced from symbol traces using `gdb`.

As shown in Figure 3, in the Trilogy implementation, the main thread contexts A, B, and C generate context switches in a stable cycle.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 November 2025 doi:10.20944/preprints202511.0596.v1

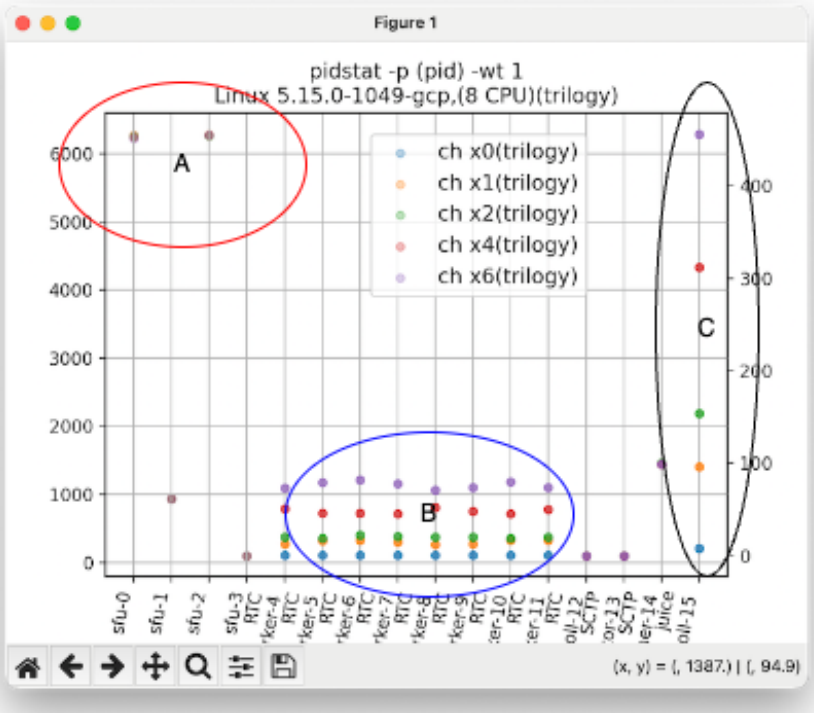

Figure 3. CS distribution for each SFU context by pidstat (Trilogy)

- **Point A** In SFU-0 (main thread) and SFU-2 (DB Ingress context), a polling loop with an interval of 1 ms as designed is maintained, and stable scheduling was confirmed even under increased load.
- **Point B** The RTC / SCTP / JUICE prefixes are internal threads of `libdatachannel`, and at PeerConnection initialization, `std::threads` equal to the number of CPU cores are generated, and event wait loops are resident.
- **Point C** The JUICE lower layer (UDP socket processing) uses a receive loop based on `poll(2)`, and showed a tendency for CS to increase in proportion to the number of sessions.

As shown in Figure 4, the number of `poll(2)` calls and the presence or absence of `ppoll(2)` usage were confirmed as the main differences between the two implementations.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 11 November 2025 doi:10.20944/preprints202511.0596.v1

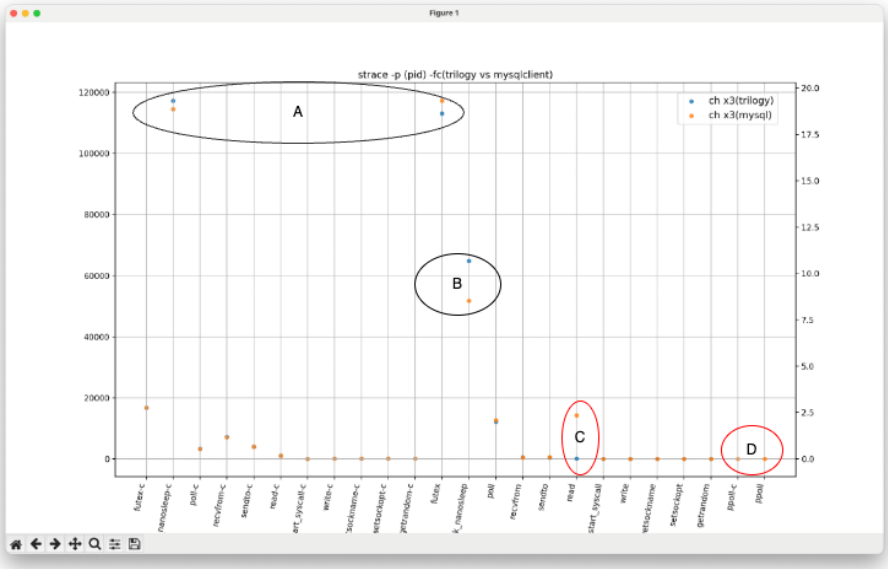

Figure 4. System call comparison by `strace` (Trilogy vs mysqlclient)

- **Point A/B** The difference between Trilogy and mysqlclient appeared as a difference in lock waits and the number of `usleep(2)` calls.
- **Point C/D** `mysqlclient` uses `ppoll(2)` and performs waits accompanied by signal mask processing. On the other hand, Trilogy specifies infinite waiting (`timeout=-1`) with `poll(2)` only, and waits for socket state changes while occupying the thread context.

Listing 1: Socket wait implementation example in Trilogy

```
static int _cb_wait(trilogy_sock_t *_sock, trilogy_wait_t wait)
{
    struct pollfd pfd = {.fd = trilogy_sock_fd(_sock)};
    while (1) {
        int rc = poll(&pfd, 1, -1); /* infinite wait */
        if (rc < 0) {
            if (errno == EINTR) continue;
            return TRILOGY_SYSERR;
        }
        return TRILOGY_OK;
    }
}
```

The design of Trilogy does not aim to improve throughput by asynchronous I/O, but aims at **maintaining the performance of the main context by reducing context switches.** By performing infinite waiting with `poll(2)`, it can wait while keeping the CPU cache, and suppress unnecessary thread scheduling. In contrast, in the `libmysqlclient` implementation (`vio/viosocket.cc`), it was confirmed that 0 (non-blocking) is specified for the timeout of `poll(2)`. This is as shown in the following excerpt.

Listing 2: poll call in mysqlclient

```
int poll_one(Socket socket, Poll_mode mode, bool wait, uint64_t timeout_usec)
{
  struct pollfd fds = {};
  fds.fd = socket;
  int timeout =
    !wait ? 0
    : timeout_usec > 0 ? static_cast<int>((1000+timeout_usec) / 1000) : -1;
  return ::poll(&fds, 1, timeout);
}
```

 doi:10.20944/preprints202511.0596.v1

As a result of `gdb` analysis, in the `mysqlclient` implementation, non-blocking operation by `poll(0)` was frequently observed, and since this is combined with `ppoll(2)` accompanied by signal mask processing, the number of context switches tends to be larger than Trilogy.

3.5. Discussion

The reduction of context switches in Trilogy is based on a design that suppresses competition for CPU caches and memory access bandwidth, and maintains consistent performance of the application main context. In modern multi-core environments, it suggests that suppressing unnecessary scheduling contributes more to performance improvement than improving throughput through asynchronous I/O.

On the other hand, the design that causes frequent context switches by using short-timeout polling and signal masks as in `libmysqlclient` is effective in ultra-high-load and multi-connection environments. Because frequent scheduling distributes CPU usage and ensures fairness among waiting threads, the balance between fairness and throughput is improved. In a single IPC environment such as this study, rather, suppressing context switches by infinite waiting with `poll(-1)` contributes to maintaining main context performance, but in multi-connection environments, the strategy of signal mask + short-term polling tends to be advantageous.

4. Related Works

Regarding performance design and communication efficiency in microservice architectures, various studies have been conducted in recent years. The study "Experimental Evaluation of Architectural Software Performance Design Patterns in Microservices" by Meijer, Trubiani, and Aleti (2024) [3] is a comprehensive research work that experimentally evaluated the impact of microservice design patterns on system performance. In that study, three design patterns—Gateway Aggregation, Gateway Offloading, and Pipes and Filters— were targeted, and measurements were collected in actual cloud environments and consistency with the theoretical model (queueing network model) was verified. As a result, the differences between theoretical performance prediction and measured results in actual operation were clarified, particularly the occurrence of bottleneck switching and nonlinear behavior of CPU utilization. The study by Meijer et al. clarifies the performance impact of design choices across the entire microservice configuration, while this study is characterized by quantifying the specific impact of implementation choices in the communication layer (whether `libmysqlclient` or Trilogy) on system performance. While the former discusses macro-level optimization of design patterns, the latter (this study) presents micro-level optimization of the library layer that forms its basis, and the two are in a complementary relationship.

As a future prospect, by integrating the IPC optimization knowledge at the library level obtained in this study (such as CS reduction effects and asynchronous I/O control characteristics) into performance models at the design-pattern level, it may be possible to construct a more accurate performance prediction model for the entire microservice architecture.

4.1. Context Switch and Performance Degradation in ManyCore Environments

The effectiveness of context switch (CS) reduction focused on in this study is a particularly important issue in modern multi-core and many-core architectures. In multi-core CPU environments, it is known that frequent CS between threads causes significant performance degradation due to the following factors.

- **Loss of cache locality.** Every time a CS occurs, the CPU cache lines are flushed, and the working set of the next thread is reloaded. This increases the L1/L2 cache miss rate and makes memory access latency significant.
- **Movement between NUMA (Non-Uniform Memory Access) nodes.** When a thread is scheduled to a different core, the memory reference node changes, increasing memory latency.

 doi:10.20944/preprints202511.0596.v1

- **Increase in interrupt handlers and kernel lock contention.** In environments with high CS frequency, contention for kernel locks increases, and CPU cycles are wasted due to spinlocks.

These phenomena are particularly serious in microservice platforms that employ high-thread-density IPC processing and highly parallel event loops, and it has been reported that stabilization strategies by CS reduction are effective.

4.2. Summary

In modern many-core (16–64 core) environments: The Trilogy-type is optimal for use cases that "pin the CPU to a small number of tasks" (for example, low-latency IPC). The mysqlclient-type is optimal for use cases that "rotate the CPU among many tasks" (for example, web servers or DB clients).

Therefore, which is superior depends on the use case. When MySQL is used as IPC together with CPU pinning, the Trilogy-type is better (to dominate the main loop with `poll(-1)`), whereas in cases such as MySQL Connector or DB Proxy, which handle hundreds of connections in one process without CPU pinning, the strategy of frequent CS (short `poll`/`ppoll`) is considered to be globally optimal.

References

1. Oracle Corporation: MySQL Internals Manual, 2024. https://dev.mysql.com/doc/internals/en/
2. GitHub: Trilogy is a client library for MySQL-compatible database servers, designed for performance, flexibility, and ease of embedding., 2023. https://github.com/trilogy-libraries/trilogy
3. W. Meijer, C. Trubiani, A. Aleti: "Experimental Evaluation of Architectural Software Performance Design Patterns in Microservices," *Journal of Systems and Software*, 2024. DOI: 10.1016/j.jss.2024.112183. https://doi.org/10.1016/j.jss.2024.112183