Technical Note

A Low-Overhead Inter-Process Communication Library with Minimal Dependencies for Efficient Microservice Communication

Daisuke Sugisawa *

Posted Date: 31 December 2025

doi: 10.20944/preprints202511.0596.v3

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

 doi:10.20944/preprints202511.0596.v3

Technical Note

A Low-Overhead Inter-Process Communication Library with Minimal Dependencies for Efficient Microservice Communication

Daisuke Sugisawa

Xander, LLC. Shibuya, Tokyo, Japan; daisuke.sugisawa.xander@gmail.com

Abstract

In the modern microservice environment, library dependencies 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 verified through long-term operation.

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 2). This structure makes it difficult to apply lightweight or embedded environments.

In contrast, the lightweight MySQL client library **Trilogy** [3] 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) (This implementation has been published as a Trilogy Pull Request. (commit hash: a61c97a) [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.v3

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. Table 1 shows the list.

Table 1. Source files required for replication mechanism.

	Source File	Function
1	sql/log_event.cc	Binlog event generation and base class definition
2	sql/rpl_utility.cc	Common utilities for replication
3	sql/rpl_gtid_tsid_map.cc	GTID/TSID management
4	sql/rpl_gtid_misc.cc	GTID auxiliary functions
5	sql/rpl_gtid_set.cc	GTID set operation
6	sql/rpl_gtid_specification.cc	GTID definition structure
7	sql/rpl_tblmap.cc	Table Map event processing
8	sql/basic_istream.cc	Basic implementation of stream I/O
9	sql/binlog_istream.cc	Binlog input stream processing
10	sql/binlog_reader.cc	Binlog reader class
11	sql/stream_cipher.cc	Binlog encryption processing
12	sql/rpl_log_encryption.cc	Replication log encryption
13	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 2 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:** A large number of dependencies reduces reproducibility, 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 2). 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 2. 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

 doi:10.20944/preprints202511.0596.v3

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

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 a lightweight and loosely coupled IPC layer (Figure 1).

- `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

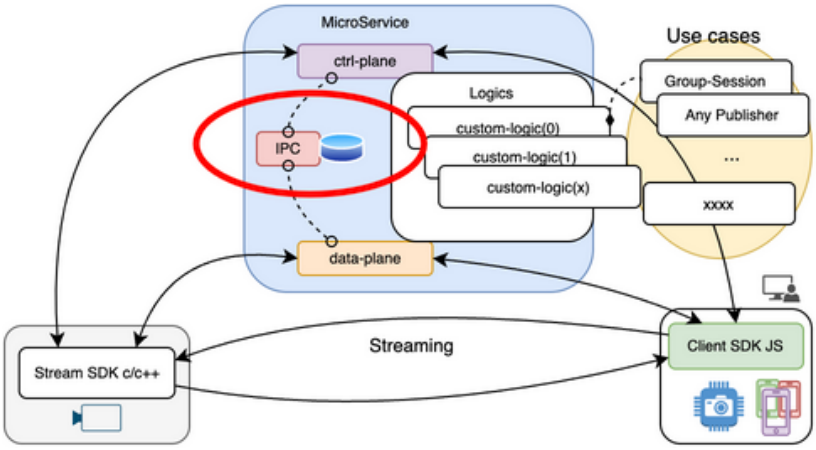

Figure 1. IPC session.

Table 3. mysql Trilogy Dependencies.

Item	Conventional (libmysqlclient)	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

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.

 doi:10.20944/preprints202511.0596.v3

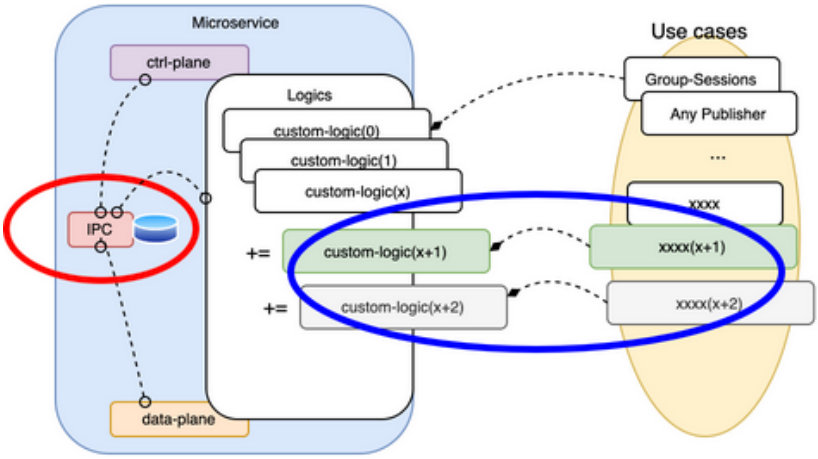

Figure 2. IPC pub-sub.

3.3. Comparison of Evaluation Results

In this section, we compare **operational metrics** for communication between the control plane and data plane in SFU when using the proposed method (Trilogy extension) and the conventional configuration (`libmysqlclient`). Both configurations were implemented on a Docker environment, and CPU noop (halt) time during IPC communication associated with SDP exchange processing was measured. The HALT value is an indicator of the CPU processing capacity remaining per processing cycle after the Edge-SFU performs real-time packet processing, and is an indicator that comprehensively reflects I/O wait, packet throughput, number of remaining processes in the loop, etc. By using this indicator, node load and the accuracy of autonomous control decisions under actual operation can be quantitatively evaluated. In addition, container size and build time were also measured, and the difference in overall build cost and runtime overhead was confirmed. The evaluation results are shown in Table 4.

Table 4. Comparison between Trilogy and libmysqlclient (Docker environment)

Item	Trilogy	mysql client	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)	We varied the number of accommodated sessions from 1 to 8 and evaluated the statistics during actual SFU operation, but no significant difference was confirmed.

In the Trilogy-based configuration, significant improvements were confirmed in build cost, dependent libraries, and container size compared to the `libmysqlclient` configuration. In this study, by selecting a simple INSERT-ONLY workload that does not consider transactional consistency or multi-statements for the purpose of stable delivery for pub/sub use, **(a simple setup that does not consider transactional consistency or complex lock contention)** the evaluation became dominated by scheduler behavior rather than being I/O-bound.

 doi:10.20944/preprints202511.0596.v3

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, main thread contexts A, B, and C exhibited stable context-switch cycles.

- **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` [2], 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)` or `ppoll(2)`, and showed a tendency for CS to increase in proportion to the number of sessions.

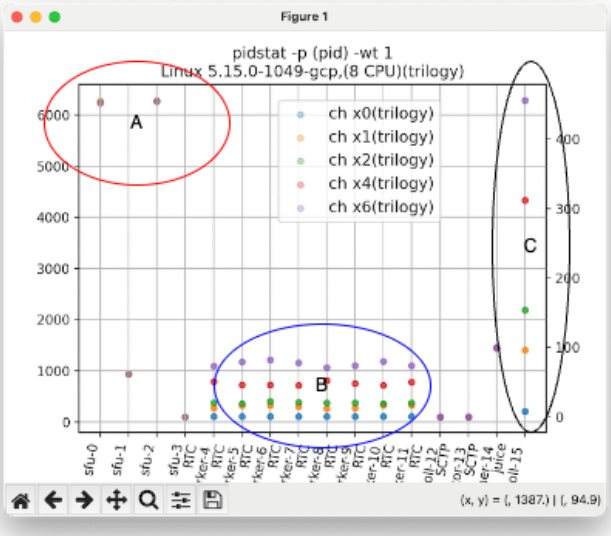

Figure 3. CS distribution for each SFU context by pidstat (Trilogy)

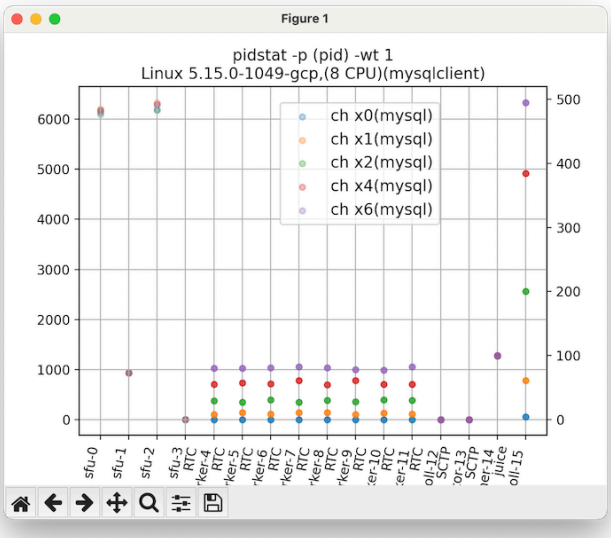

Figure 4. CS distribution for each SFU context by pidstat (mysqlclient)

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 31 December 2025 doi:10.20944/preprints202511.0596.v3

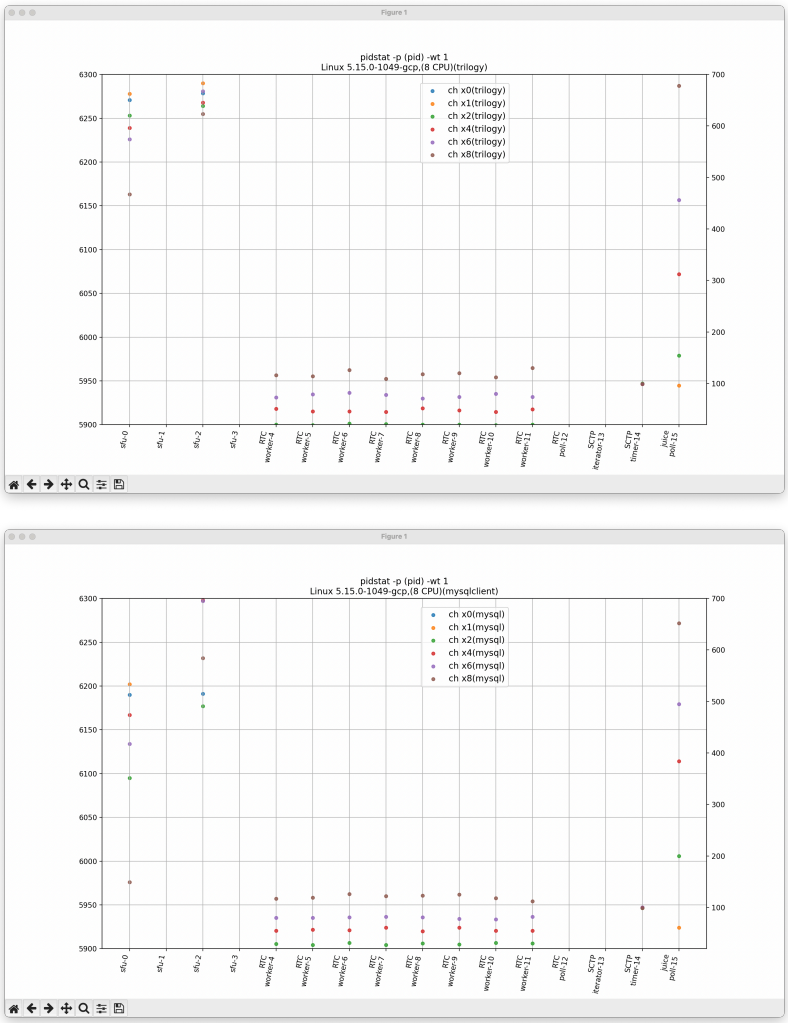

Figure 5. CS distribution for each SFU context by pidstat (mysqlclient detail)

As shown in Figure 6, it was confirmed that the call arguments (0 or -1) of `poll(2)`, the number of calls, and whether or not `ppoll(2)` is used are the main differences between the two implementations.

- **Point A/B** As shown in Figure 6, the difference between Trilogy and mysqlclient appeared as a difference in lock waits and the number of `usleep(2)` calls.
- **Point C/D** As shown in Figure 6, mysqlclient uses `poll(pfd, 1, 0);`, whereas Trilogy specifies infinite waiting (`timeout=-1`) with `poll(pfd, 1, -1);`, 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;
    }
}
```

 doi:10.20944/preprints202511.0596.v3

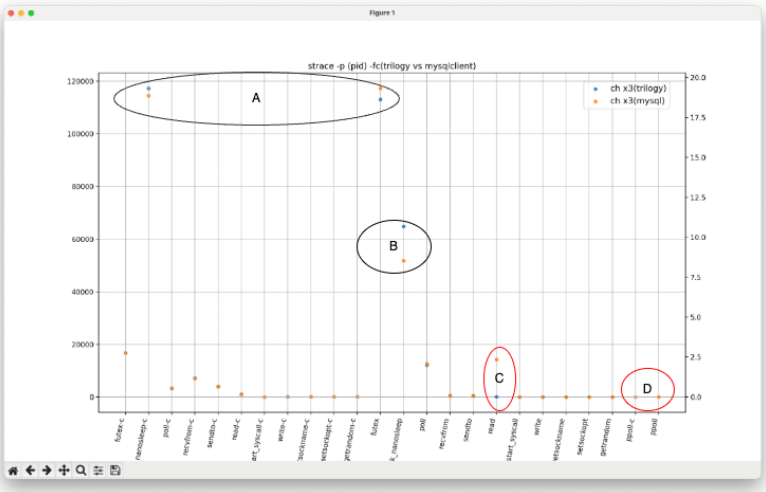

Figure 6. System call comparison by `strace` (Trilogy vs mysqlclient)

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);
}
```

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. This function is implemented in MySQL Connector CPP (mysql/mysql-connector-cpp/blob/trunk/cdk/foundation/socket_detail.cc#L862), and in the path linked as libmysqlclient, timeout_usec=0 is passed as a fixed value. [2] In the internal implementation of MySQL Connector/C (libmysqlclient), the timeout of poll() is fixed to 0 in the normal path, and no means of changing it from the application side is provided. Although there are some paths that specify poll(-1) or arbitrary timeout in connection establishment and TLS processing, in persistent stream processing such as COM_BINLOG_DUMP, poll(0) is used fixedly. Therefore, in the normal path, it should be regarded as a "design policy difference" rather than a "default difference."

3.5. Discussion

poll(0): User-space loop (busy-ish) → increases context switches, causes non-linear spikes.

poll(-1): Kernel manages sleep → stable RUNNABLE density, easier cooperation between scheduler and IRQ.

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

[2] https://github.com/mysql/mysql-connector-cpp/blob/trunk/cdk/foundation/socket_detail.cc

 doi:10.20944/preprints202511.0596.v3

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 as in `libmysqlclient` is effective in 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 some use cases. In a single IPC environment such as this study, suppressing context switches by infinite waiting with `poll(-1)` contributes to maintaining main context performance, but in multi-connection environments, the signal mask + short-term polling strategy may be advantageous.

4. Limitations and Future Work

This report focuses on the waiting strategies of `poll(0)` or `poll(-1)` under the CFS scheduler (Completely Fair Scheduler), but in recent years, `io_uring Linux 5.1` has been introduced to asynchronously control kernel I/O from user space. Kernel-bypass designs such as io_uring and DPDK aim to eliminate scheduler intervention, which differs from the "waiting optimization based on scheduling behavior" in this paper. Therefore, they are excluded from the comparison in this study.

5. Related Works

5.1. Microservice Design Patterns

Meijer et al. [4] experimentally evaluated the impact of microservice design patterns (Gateway Aggregation, etc.) on system performance, and clarified bottleneck transitions and nonlinear behavior in CPU utilization. This study is complementary in that it presents micro-level optimization at the library layer (libmysqlclient vs Trilogy) in contrast to macro-level optimization of design patterns.

5.2. Linux Scheduler's Complex Load Balancing Algorithms

Lozi et al. [6] showed that Linux scheduler's complex load balancing algorithms cause unnecessary thread waiting in NUMA environments. The `poll(-1)` strategy in the Trilogy implementation in this study is an approach to realize scheduler simplification from the application layer by suppressing excessive CS.

5.3. poll, ppoll

In multi-core CPU environments, frequent CS causes the following performance degradation [5]: (1) increased L1/L2 cache miss rate due to loss of cache locality, (2) increased memory latency due to NUMA node migration, (3) spinlock waiting due to kernel lock contention. These are particularly serious in high-thread-density IPC processing.

5.4. Performance Characteristics

The effectiveness of reducing context switches, which is the focus of this study, is a particularly important issue in modern multi-core and many-core architectures. In multi-core CPU environments, frequent CS between threads is known to cause significant performance degradation due to the following factors.

- **Loss of cache locality** Every time a CS occurs, CPU cache lines are flushed and the working set of the next thread is reloaded. This increases the L1/L2 cache miss rate and causes significant memory access delays.
- **NUMA (Non-Uniform Memory Access) node migration** When threads are scheduled on different cores, the memory reference destination node changes, increasing memory latency.
- **Increased interrupt handlers and kernel lock contention** In environments with high CS frequency, kernel lock acquisition contention increases, and CPU cycles are wasted by spin locks.

 doi:10.20944/preprints202511.0596.v3

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

6. Conclusion

In modern ManyCore (≈64 core) environments, the characteristics of Trilogy-type and mysqlclient-type can be summarized as follows.

- **Trilogy-type** is optimal for "pinning CPU to a small number of tasks" use (e.g., low-latency IPC). This method uses `poll(-1)` for long-term waiting, avoiding frequent wakeups by the kernel scheduler, and suppressing unnecessary context switches. As a result, cache locality is maintained and the effective CPU utilization rate is stabilized.
- **mysqlclient-type** is optimal for "rotating CPU through many tasks" use (e.g., web servers, DB clients) This method repeats short-cycle `poll` / `ppoll`, making it easier for the CFS scheduler to ensure fairness among tasks, and can improve throughput in multi-connection environments.

Therefore, which is better depends on the use, and when "using MySQL as IPC" in combination with CPU Pinning, the Trilogy-type is suitable (`poll(-1)` dominates the main loop). Conversely, in cases such as MySQL Connector and DB Proxy, where CPU Pinning is not used and "one process handles hundreds of connections," it is considered that the CS-intensive (short `poll` / `ppoll`) strategy is more globally optimal.

The final author version of this paper has been archived at Sugisawa, D. (2025). A Low-Overhead Inter-Process Communication Library with Minimal Dependencies for Efficient Microservice Communication. Preprints.[3] The implementation source code is published on GitHub (https://github.com/trilogy-libraries/trilogy/pull/247).

Listing 3: inc/trilogy/client.h

```
/* trilogy_binlog_dump - Send a binlog dump command to the server.
 *
 * This should only be called while the connection is ready for commands.
 *
 * conn     - A connected trilogy_conn_t pointer. Using a disconnected
 *            trilogy_conn_t is undefined.
 * data     - The binlog dump request data.
 * data_len - The length of the request data in bytes.
 *
 * Return values:
 *   TRILOGY_OK     - The binlog dump command was successfully sent to the
 *                    server.
 *   TRILOGY_AGAIN  - The socket wasn't ready for writing. The caller should wait
 *                    for writeability using `conn->sock`. Then call
 *                    trilogy_flush_writes.
 *   TRILOGY_SYSERR - A system error occurred, check errno.
 */
int trilogy_binlog_dump(trilogy_conn_t *conn, const char *data, size_t data_len);

/* trilogy_binlog_dump_recv - Read a binlog event from the server.
 *
 * This should be called after all data written by trilogy_binlog_dump is
 * flushed to the network. Calling this at any other time during the connection
 * lifecycle is undefined.
 *
 * conn          - A connected trilogy_conn_t pointer. Using a disconnected
 *                 trilogy_conn_t is undefined.
 * binlog_event  - Out parameter; A pre-allocated trilogy_binlog_event_t pointer.
 *                 If TRILOGY_OK is returned, this struct will be filled out with
 *                 the binlog event data.
 *
 * Return values:
 *   TRILOGY_OK                - A binlog event was successfully read from the
 *                               server.
 *   TRILOGY_AGAIN             - The socket wasn't ready for reading. The caller
 *                               should wait for readability using `conn->sock`.
 *                               Then call this function until it returns a
```

[3] https://doi.org/10.20944/preprints202511.0596.v2

 doi:10.20944/preprints202511.0596.v3


```
 *                               different value.
 *   TRILOGY_UNEXPECTED_PACKET  - The response packet wasn't what was expected.
 *   TRILOGY_PROTOCOL_VIOLATION - An error occurred while processing a network
 *                               packet.
 *   TRILOGY_CLOSED_CONNECTION  - The connection is closed.
 *   TRILOGY_SYSERR             - A system error occurred, check errno.
 */
int trilogy_binlog_dump_recv(trilogy_conn_t *conn, trilogy_binlog_event_t* binlog_event);
```

Listing 4: inc/trilogy/protocol.h

```
typedef struct {
    uint8_t response_code;
    uint32_t timestamp;
    uint8_t event_type;
    uint32_t server_id;
    uint32_t event_size;
    uint32_t position;
    uint16_t event_flags;
    uint8_t *data;
    size_t data_len;
} trilogy_binlog_event_t;
```

Listing 5: src/protocol.c

```
int trilogy_parse_binlog_event_packet(const uint8_t *buff, size_t len, trilogy_binlog_event_t *
  binlog_event)
{
    if (!binlog_event || (len <= 20)) {
        return(TRILOGY_ERR);
    }
    trilogy_reader_t reader = TRILOGY_READER(buff, len);
    trilogy_reader_get_uint8(&reader, &binlog_event->response_code);
    trilogy_reader_get_uint32(&reader, &binlog_event->timestamp);
    trilogy_reader_get_uint8(&reader, &binlog_event->event_type);
    trilogy_reader_get_uint32(&reader, &binlog_event->server_id);
    trilogy_reader_get_uint32(&reader, &binlog_event->event_size);
    trilogy_reader_get_uint32(&reader, &binlog_event->position);
    trilogy_reader_get_uint16(&reader, &binlog_event->event_flags);
    if (binlog_event->data_len > (len - 20)) {
        for(size_t n = 0; n < (len - 20); n++) {
            trilogy_reader_get_uint8(&reader, ((uint8_t*)(binlog_event->data)) + n);
        }
    }
    return trilogy_reader_finish(&reader);
}
```

Listing 6: src/client.c

```
int trilogy_binlog_dump(trilogy_conn_t *conn, const char *data, size_t data_len)
{
    #define TRILOGY_CMD_BINLOG_DUMP 0x12
    int err = 0;
    trilogy_builder_t builder;
    err = begin_command_phase(&builder, conn, 0);
    if (err < 0) {
        return err;
    }
    err = trilogy_builder_write_uint8(&builder, TRILOGY_CMD_BINLOG_DUMP);
    if (err < 0) {
        return err;
    }
    err = trilogy_builder_write_buffer(&builder, data, data_len);
    if (err < 0) {
        return err;
    }
    trilogy_builder_finalize(&builder);
    conn->packet_parser.sequence_number = builder.seq;
    return begin_write(conn);
}

int trilogy_binlog_dump_recv(trilogy_conn_t *conn, trilogy_binlog_event_t* binlog_event) {
    int err = read_packet(conn);
    if (err < 0) {
        return err;
    }
    return trilogy_parse_binlog_event_packet(
        conn->packet_buffer.buff,
```

 doi:10.20944/preprints202511.0596.v3


```
        conn->packet_buffer.len,
        binlog_event
    );
}
```

References

1. Oracle Corporation: MySQL Internals Manual, 2024. https://dev.mysql.com/doc/internals/en/
2. Paul-Louis Ageneau, *libdatachannel*, GitHub repository, https://github.com/paullouisageneau/libdatachannel.
3. 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
4. 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
5. The Linux man-pages project: “poll(2), ppoll(2) - Wait for some event on a file descriptor,” Linux Programmer’s Manual, https://man7.org/linux/man-pages/man2/poll.2.html
6. J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quema, A. Fedorova: "The Linux Scheduler: a Decade of Wasted Cores," Proceedings of EuroSys 2016, ACM, pp. 1–16, DOI: 10.1145/2901318.2901326. https://people.ece.ubc.ca/sasha/papers/eurosys16-final29.pdf