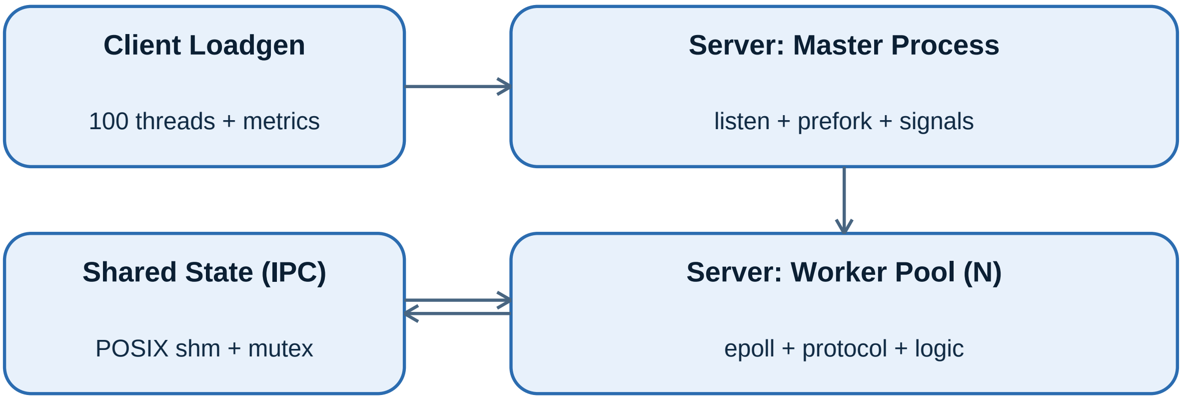
Bank Vault

High-Concurrency Client-Server Network Service System

Project Plan & Technical Design

Date: December 12, 2025 (Asia/Taipei)

Team: (Server/IPC), (Protocol/Libs), (Client/Benchmarks)



Elevator pitch: A mini core-banking service that proves OS-level mastery: prefork multi-process server, IPC-backed shared account state with process-shared locking, a custom binary protocol with CRC integrity checks (and optional XOR payload encryption), plus a multi-threaded client load generator that reports throughput and latency percentiles.

1. Goals and Requirements Map

This project implements a high-concurrency client-server system with custom protocol and fault tolerance. The design intentionally maps one-to-one to the course requirements.

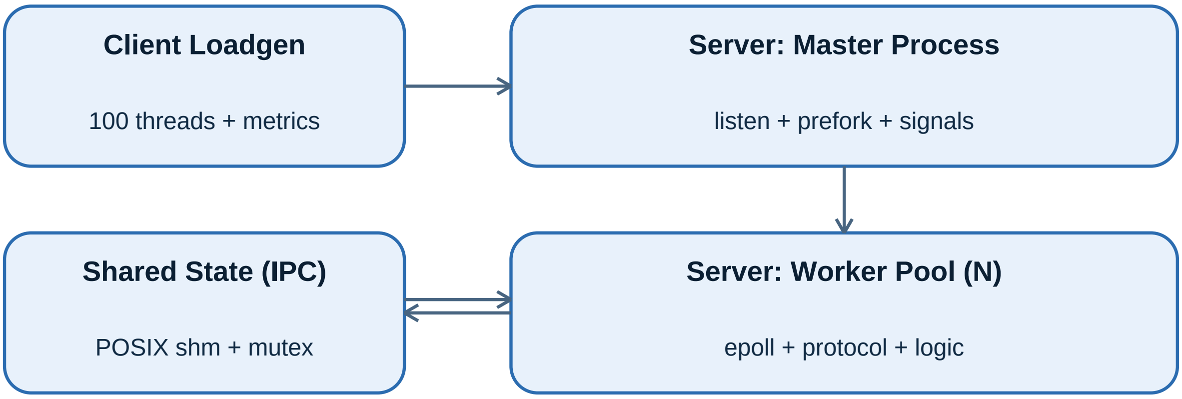
Requirement	Design choice in Bank Vault
Client: multi-threaded (>= 100 conns)	Load generator spawns 100+ threads with barrier start. Each thread drives a session (login + ops) and records latency.
Server: multi-process	Master preforks N worker processes (Master-Worker). Workers handle clients using epoll.
IPC required	POSIX shared memory stores account balances and global counters; process-shared locking ensures correctness.
Custom app protocol (no HTTP/WebSocket)	Binary framing: [Len][Magic][Ver][Flags][Op][Seq][CRC32][Body]. Opcodes implement banking operations.
Shared libraries	libnet (socket/epoll wrappers), libproto (encode/decode, CRC32, XOR), liblog (logging).
Security (>= 1)	CRC32 integrity for every packet; optional XOR encryption; authentication handshake.
Reliability (>= 1)	Heartbeat + timeouts; graceful shutdown on SIGINT; optional worker respawn on crash.

Success criteria

- Sustain 100 concurrent client connections without crashing.
- Maintain correct balances under concurrent transfers and withdrawals (no lost updates, no negative balance).
- Demonstrate measured throughput (req/s) and latency percentiles (p50/p95/p99).
- Clean shutdown releases IPC resources (no leaked shared memory objects).

2. System Overview

The system consists of a multi-threaded client load generator and a prefork multi-process server. Workers run epoll loops for scalable I/O, while IPC-backed shared state provides consistent account management across processes.



Key design decisions

- Workers share the listening socket and call accept() directly (simple prefork).
- Each worker uses epoll to handle many clients concurrently (high concurrency without one-process-per-client).
- Shared memory holds balances and counters; process-shared locking ensures data integrity.
- Protocol is length-prefixed with CRC32 for integrity and a Magic value for framing recovery.

3. Server Architecture (Multi-Process)

The server implements a Master-Worker architecture. The master sets up the listening socket, initializes IPC resources, then preforks workers. Workers handle I/O and business logic.

3.1 Process model

- Master: initializes listening socket, IPC, configuration; spawns N workers; handles SIGINT/SIGTERM; monitors worker liveness (optional respawn).
- Workers: epoll event loop; per-connection session state; protocol decode/encode; executes bank operations with shared-state locking.

3.2 Connection handling strategy

Recommended baseline (prefork accept): all workers inherit the same listening socket and call accept(). The OS load-balances connections across workers. This is standard, easy to implement, and sufficient for the course requirements.

Optional advanced version: master accepts connections and passes client fds to workers via a Unix domain socket using SCM_RIGHTS (showy, but more complex).

3.3 Worker I/O model

- Non-blocking sockets for clients; one epoll instance per worker.
- Per-client read buffer supports partial reads and reassembly via the length-prefixed frame.
- Per-client write queue supports partial writes (EAGAIN) without blocking the worker.

3.4 Error handling policy

- Malformed frames: reject with error response and close connection if repeated.
- CRC failure: respond with ERR_BAD_CRC; optionally request retransmit; close on abuse.
- Server busy: respond with ERR_BUSY (rate limit) or apply backpressure by pausing read events.

4. IPC and Shared State

IPC is used to synchronize account balances and global service statistics across worker processes. The baseline uses POSIX shared memory with a process-shared pthread mutex.

4.1 POSIX shared memory layout

```
// Stored in shm (mmap). All workers attach read-write.
#define MAX_ACCOUNTS 10000

typedef struct {
pthread_mutex_t global_lock; // PTHREAD_PROCESS_SHARED
uint64_t total_requests;
uint64_t total_errors;
uint32_t active_connections;
uint32_t shutdown_flag; // set by master on SIGINT
struct {
pthread_mutex_t lock; // per-account lock (process-shared)
int64_t balance_cents; // store cents to avoid floating bugs
} acct[MAX_ACCOUNTS];
} vault_shm_t;
```

4.2 Locking rules (data consistency)

- Deposit/Withdraw: lock the account, update balance, unlock.
- Transfer: lock both accounts in a stable order (min acct id first) to prevent deadlocks.
- Counters: protected by global_lock or atomic operations (optional).

4.3 Crash safety note

If a worker crashes while holding a mutex, the system can stall. As a mitigation, enable robust mutexes (pthread_mutexattr_setrobust) and handle EOWNERDEAD by marking shared state consistent again. This is optional but demonstrates advanced OS knowledge.

5. Custom Protocol Design

The protocol is a binary, length-prefixed application-layer protocol with integrity checks. It is not HTTP/WebSocket.

5.1 Frame format

Field	Size	Description
Length	4 bytes (uint32)	Total packet length in bytes (header + body). Network byte order.
Magic	2 bytes	0xC0DE. Used for framing recovery and quick validation.
Version	1 byte	Protocol version (start at 1).
Flags	1 byte	bit0: XOR encrypted body, bit1: error response, bit2: reserved.
OpCode	2 bytes (uint16)	Operation ID (login, deposit, withdraw, ...).
Seq	4 bytes (uint32)	Client-chosen sequence number for matching responses.
CRC32	4 bytes (uint32)	CRC32 over header (with CRC=0) + body.
Body	variable	OpCode-specific payload.

5.2 OpCode set

OpCode	Name	Request Body	Response Body
0x0001	LOGIN	user_len(1)+user, pass_len(1)+pass	status(2), session_id(4)
0x0002	DEPOSIT	acct_id(4), amount_cents(8)	status(2), new_balance(8)
0x0003	WITHDRAW	acct_id(4), amount_cents(8)	status(2), new_balance(8)
0x0004	TRANSFER	from(4), to(4), amount_cents(8)	status(2), from_balance(8), to_balance(8)
0x0005	BALANCE	acct_id(4)	status(2), balance(8)
0x00F0	PING	(empty)	status(2), server_time_ms(8)

5.3 Session key and optional XOR encryption

After LOGIN succeeds, the server returns a session_id. A simple session key can be derived (e.g., key = CRC32(user||session_id)) and used to XOR the Body bytes when the encrypted flag is set. This keeps the project within scope while meeting the encryption requirement.

6. Security and Reliability Features

The project includes an integrity check (CRC32) for every packet and implements reliability features to handle disconnections and clean shutdown.

6.1 Security

- Integrity: CRC32 validates each frame; packets failing CRC are rejected.
- Authentication: LOGIN handshake required before money operations.
- Encryption (optional): XOR-encrypted payload using a session-derived key (flag-controlled).

6.2 Reliability

- Heartbeat: clients send PING periodically; server drops idle connections after timeout.
- Timeout handling: client sets read/write timeouts and retries idempotent operations (BALANCE) when safe.
- Graceful shutdown: SIGINT triggers shutdown_flag; workers stop accepting, finish in-flight requests, close fds, then detach and unlink shared memory.
- Worker crash isolation: a crashed worker only drops its clients; other workers continue. Optional master respawn.

7. Client Load Generator and Metrics

The client is a multi-threaded stress-testing tool designed to open 100+ concurrent sessions, execute randomized banking operations, and report throughput and latency statistics.

7.1 Threading model

- Thread-per-connection baseline: each thread maintains one TCP connection and sends sequential requests.
- Barrier start: all threads begin load at the same time to generate concurrency spikes.
- Configurable run length: operations per thread, distribution of opcodes, and target accounts.

7.2 Metrics output (example)

```
vault_client --host 127.0.0.1 --port 7777 --threads 100 --ops 2000 --mix
transfer=10,withdraw=20,deposit=30,balance=40

RESULT:
total_ops=200000 ok=199872 err=128 duration=12.34s
throughput=16206.8 req/s
latency_ms: p50=1.2 p95=4.8 p99=9.7 max=41.3
reconnects=3 timeouts=11
```

7.3 Correctness checks

- Invariant: balances never become negative (server enforces).
- Transfer atomicity: after transfer, sum of balances across involved accounts changes only by fees (if any) otherwise unchanged.
- End-to-end: compare server-side totals to client-side expected totals under controlled test cases.

8. Repository Layout and Build System

The project uses a Makefile-based build and shares common code via libraries to meet the modularity requirement.

8.1 Suggested repo layout

```
repo/
include/ # public headers shared by client + server
libnet/ # socket wrappers, epoll helpers
libproto/ # framing, CRC32, (optional) XOR
liblog/ # logging, rotating files
server/ # master-worker server, shm init, business logic
client/ # multi-thread load generator + stats
tests/ # unit/integration tests
Makefile
README.md
```

8.2 Make targets

- make all: builds libs + server + client
- make server: builds vault_server
- make client: builds vault_client
- make test: runs unit tests (protocol + CRC) and integration tests (single worker + scripted ops)
- make clean: removes build artifacts

8.3 Runtime commands (example)

```
# server: prefork 4 workers, 10k accounts
./vault_server --port 7777 --workers 4 --accounts 10000

# client: 100 concurrent connections, mixed ops
./vault_client --host 127.0.0.1 --port 7777 --threads 100 --ops 2000
```

9. Testing Plan, Milestones, and Demo Checklist

9.1 Testing plan

- Unit tests: protocol encode/decode, framing recovery, CRC32 correctness, XOR roundtrip.
- Concurrency tests: 100 threads, random transfers; validate invariants (no negative, atomic transfer).
- Failure tests: drop connections, delay responses, send bad CRC, send truncated frames; verify robust handling.
- Shutdown test: SIGINT during load; confirm resources cleaned (shm_unlink) and workers exit cleanly.

9.2 Milestones (2-3 week schedule)

Week	Deliverables
W1	Protocol framing + libproto + skeleton server/client (connect, echo, CRC).
W2	Prefork server + epoll worker loop + shared memory + locking + core ops (deposit/withdraw/balance).
W3	Transfers + authentication + heartbeat + graceful shutdown + benchmarks + README + screenshots.

9.3 Team role split (example)

- Member A: server core (prefork, epoll loop, session handling, graceful shutdown).
- Member B: IPC + locking + shared memory design + correctness tests.
- Member C: protocol libs + client load generator + latency/throughput reports.

9.4 Demo + screenshot checklist

- Screenshot: server running with multiple worker processes (ps output).
- Screenshot: client run showing throughput and latency percentiles.
- Screenshot: CRC failure handling (client sends bad packet -> server rejects).
- Screenshot: SIGINT graceful shutdown and IPC cleanup proof (no lingering shm object).

Appendix A. Status and Error Codes

Code	Name	Meaning
0	OK	Success
1	ERR_AUTH	Not logged in or login failed
2	ERR_BAD_CRC	Integrity check failed
3	ERR_BAD_FRAME	Malformed or unsupported packet
4	ERR_NO_ACCOUNT	Account id out of range
5	ERR_INSUFFICIENT	Not enough balance to withdraw/transfer
6	ERR_BUSY	Server overloaded or rate-limited
7	ERR_TIMEOUT	Request timed out