초기 서버 테스트 항목과 실행

개요

게임 서버의 구조, 프로토콜, 언어와 플래폼은 게임의 특성에 영향을 받기 때문에 게임마다 크고 작은 차이가 생긴다. 또한, 해당 프로젝트를 진행하는 서버 프로그래머의 경험과 취향에 영향을 받는 부분도 크기에 하나의 서버 프레임워크로 모든 게임에 적용하기 어려운 면도 있다.

이외의 다른 여러가지 다양한 요인으로 인해 게임 서버를 구현할 때 처음부터 구현하거나 라이브에서 오래 검증되지 않은 여러 기술 요소와 선택을 갖고 개발하는 경우들이 있을 수 밖에 없다.

이렇게 여러 선택과 새로운 구현을 포함하는 게임 서버가 라이브 환경에서 사용자들이 몰리는 론칭 시점을 포함하여 안정적으로 서비스 가능한 지를 검증하는 작업이 부하 테스트이다.

부하 테스트는 예상 되는 동접 수치를 넘어서는 부하를 투입하여 서버가 문제 없이 동작하는 지 확인하는 과정을 게임 클라이언트와 비슷한 가상 클라이언트로 테스트를 진행한다.

부하 테스트 클라이언트를 통해 패킷 기반의 어뷰징이나 핵 개발을 통한 보안성 검증, 잘못된 입력 값이나 프로토콜 변경 등을 통한 안정성 검증에 함께 활용할 수 있기 때문에 출시 전 성능, 안정성, 확장성, 보안을 확인하고 대처하기 위한 가장 확실한 방법이다.

하지만, 부하 테스트는 주로 출시 전에 전체 게임 기능을 대상으로 진행하기 때문에 이 때 심각한 문제들이 발견되거나 하면 수정할 시간이 부족하거나 대처하기가 매우 곤란한 경우도 발생한다.

따라서, 좀 더 일찍 단위 테스트를 포함한 부하 테스트를 진행한다면 초기에 여러 가지 선택을 좀 더 확고하게 할 수 있기 때문에 리스크를 줄일 수 있고, 보다 안정적인 서버 기반을 갖고 있다면 기반에서 발생하는 문제를 개발 중에 겪지 않기 때문에 게임 개발 자체도 빠르게 진행할 수 있다.

테스트 항목

통신 (Communication)

게임 서버의 기반 중 기반으로 생각보다 많은 게임 서버들이 통신 구현에서 성능 문제나 오류, 암호화의 잘못된 선택, 시리얼라이제이션 방식이 게임에 맞지 않거나 하는 경우들이 있었다.

단위 테스트와 부하 테스트를 처리 구조와 함께 묶어서 초기에 테스트를 하면 대부분의 경우 사전에 문제를 제거할 수 있다.

통신 구현은 구현된 클래스 별로 단위 테스트를 진행하고 이 때 단위 성능까지 확인하는 방식으로 검증할 수 있다. 외부 라이브러리도 동일하게 단위 테스트와 단위 성능 테스트를 진행해 두면 확신을 갖고 사용할 수 있다.

처리 모델 (Processing Model)

처리 모델은 게임 월드를 분할 하는 방식을 비롯한 분산 처리를 포함하고
하나의 프로세스 안에서 쓰레드 별 처리 구조를 포함한다.

프로세스 별 처리에서 가장 많은 처리 방식은 단일 쓰레드로 여러 프로세스를
분산하여 처리하는 방식과 여러 쓰레드를 두고 락을 잡으면서 처리하는 방식이다.

이는 메세지 패싱과 분산 처리를 기반으로 하는 것과 상태 공유 멀티 쓰레드를 사용하는
방식으로 매우 오래된 두 가지 처리 모델로 회사 내 게임들도 뮤, 메틴1/2, 썬이 분산 방식이고
R2/R2M/아크로드가 상태 공유 멀티 쓰레드 방식을 사용한다.

웹 서버 기반의 게임들의 경우 웹 앱의 특성을 갖기 때문에 다른 구조를 갖게 되나
기본적으로 분산 처리를 따르고 DB나 레디스 캐시에 정보를 저장하고 stateless 하게
동작하는 구조를 갖는다.

이 외에도 세부적으로는 파이버 기반 코루틴, async / await 사용 등 다양한 선택이
있을 수 있다.

처리 모델은 단위 테스트가 가능한 부분도 있으나 전체를 검증하기 어렵기 때문에
통신과 DB 처리를 묶어서 가상 클라이언트로 부하 테스트를 진행해야 확인할 수 있다.
이미 갖고 있는 부하 테스트 툴 안에서 로그인이나 캐릭터 생성과 게임 진입 정도로
부하를 주고 확인하는 작업을 효율적으로 진행할 수 있으며, 이 때 얻은 결과로
개선 과정을 수행하여 기반을 확고히 할 수 있다.

저장소 (RDBMS, 레디스, NoSQL 등)

초기에 선택한 DBMS와 DB 사용 API와 라이브러리의 성능과 안정성을 확인하는 과정을 밟는다.

단위 테스트와 단위 성능 테스트가 대부분 가능하고, 처리 모델의 테스트를 위한
테스트에 포함하여 부하 테스트를 함께 진행할 수 있다.

테스트 실행

- 코드 공유
- 통신 단위 테스트
- 통신 단위 성능 테스트
- 부하 툴 개발
    - 프로토콜 구현
    - 부하 시나리오 구현
- 처리 모델 부하 테스트
    - DB 호출을 포함
    - 긴 시간 (하루나 이틀 정도) 실행을 포함

테스트 대상 게임마다 다르겠으나 부하 툴 프레임워크가 준비되어 있고
여러 프로토콜의 테스트 코드도 함께 포함되어 있어 대부분
1~2개월 작업하면 확인 가능할 것으로 판단한다.

핵심 과제와 기술 요소 구현의 검증

통신, 처리, 저장소가 가장 중요한 요소이나 게임별로 갖게 되는 핵심 과제로 인해
초기에 확인해야 하는 기술 요소들이 있다.

예를 들어, 대규모 심리스 서버라면 길찾기와 충돌 처리를 포함한 이동과 판정이 잘 동작하는 지 확인해야 하고, 웹 기반 서버라면 분산 처리와 캐싱이 잘 동작하는 지 확인해야 한다.

이런 요소에 대해서는 스튜디오와 협의하여 최대한 이른 시점에 함께 테스트를 통해 확인하는 과정이 필요하다. 부언하면, 부하 테스트에서 진행하게 되면 너무 늦은 시점이 될 가능성이 높기 때문이다.

테스트 예시

TCP 여러 세션의 에코 성능 측정

여러 세션에 대해 어느 정도의 성능을 TCP 구현이 제공할 수 있는 지 단위 테스트 형식으로 성능 테스트를 진행한 예시이다.

```
TEST_CASE("multiple session echo")
{
constexpr std::size_t test_count = 1;
const int test_duration = 1000;

rx::multi_thread_worker worker1(8);
worker1.start();

test_server server(&worker1);
server.listen(9999);

auto& bm = tqa::benchmark::instance();

std::vector<std::shared_ptr<test_client>> clients;

for (std::size_t i = 0; i < test_count; ++i)
{
    clients.push_back(std::make_shared<test_client>(&worker1, 256));
}

asio::ip::tcp::endpoint endpoint(
    asio::ip::address::from_string("127.0.0.1"), 9999);

bm.mark_begin("multiple_client_echo");

for (std::size_t i = 0; i < test_count; ++i)
{
    clients[i]->connect(endpoint);
}

std::this_thread::sleep_for(std::chrono::milliseconds(test_duration));

std::size_t total_echos = 0;

for (std::size_t i = 0; i < test_count; ++i)
{
    total_echos += clients[i]->receive_count();
}

bm.mark_end("multiple_client_echo", static_cast<int>(total_echos));

worker1.stop();
```

```
}
```

테스트 구현은 단순하며 다음과 같은 중요한 결과를 얻었다.

- 한 세션에서 연결 시 128개를 전송하고 동시에 에코 진행
- 128 동시 에코, 초당 92만 PPS
- 256 동시 에코, 초당 112만 PPS

위 결과를 얻기까지 구현의 오류와 개선을 반복했다. 특히, asio의 io_context가 소켓 처리에 여러 개 있으면 느려진다는 점을 발견했고 이는 테스트를 하지 않으면 미리 알기가 매우 어려운 내용이다.

타이머 처리 성능 측정

asio의 timer 처리 성능을 측정하기 위해 단위 테스트 예시이다.

```
TEST_CASE("steady timer. pooling effect")
{
std::thread::id tid, tid1, tid2;
std::chrono::steady_clock::time_point tp, tp1;
std::atomic<int> value = 0;
std::mutex mutex;
std::condition_variable cv;

{
    std::lock_guard lock(mutex);
    value = 0;
}

auto& bm = tqa::benchmark::instance();

bm.mark_begin("steady_timer_all_pool");

rx::multi_thread_worker worker(8);
worker.start();

tid = std::this_thread::get_id();
tp = std::chrono::steady_clock::now();

bm.mark_begin("steady_timer_create_pool");

for (int i = 0; i < test_count; ++i)
{
    worker.steady_timer().once("test/once", i + 1, [&] {
    std::lock_guard lock(mutex);
    tid1 = std::this_thread::get_id();
    tp1 = std::chrono::steady_clock::now();
    ++value;
    }, std::chrono::milliseconds(100));
}

bm.mark_end("steady_timer_create_pool");

while (value < test_count)
{
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
}
```

```
bm.mark_end("steady_timer_all_pool", test_count);

worker.stop();
}
```

위 테스트를 통해 다음과 같은 결과를 얻었다.

- 8 thread, 1백만 once, 1.42, 70만/초, cpu 90%
- 생성 시간이 대부분을 차지. 1.42초 걸림.

타이머 생성 시간이 오래 걸린다는 점을 확인하고 풀링을 추가하여
초당 6만 건을 더 처리할 수 있게 되었다.

서버 부하 테스트

서버에 연결, 로그인하고 종료하는 매우 단순한 테스트이다.

```
{
  "name" : "tc_login",
  "desc" : "로그인 테스트",
  "service": {
        "use_libsodium": false,
        "account_prefix": "tc",
        "password_prefix": "d3",
        "bot_count": 1000,
        "bot_start_index": 1,
        "worker_count": 8,
        "max_lock": 10,
        "log": {
      "level": "debug",
            "console": {
                "enabled": true
            },
      "forwarder": {
        "enabled" : true,
        "level" : "warn"
      }
        },
        "data_folder": "./.",
        "report": {
      "prefix": "tc_simple"
    },
    "forwarder" : {
      "listen" : "0.0.0.0:7090",
      "backlog" : 10
    }
    },
  "flow" : [
    {
      "type" : "act_delay",
      "name" : "delay_begin",
      "delay" : 1000,
      "mark" :  "mark_1",
      "slots" : {
        "failure" : { "command" :  "exit" },
        "success": { "command":  "next" }
```

```
      }
    },
    {
      "type": "act_connect",
      "name": "connect",
      "address": "10.103.36.11:8901",
      "slots": {
        "failure": { "command": "exit" },
        "success": { "command": "next" }
      }
    },
    {
      "type": "act_login",
      "name": "login",
      "slots": {
        "failure": { "command": "exit" },
        "success": { "command": "next" }
      }
    },
    {
      "type" : "act_delay",
      "name" : "delay_end",
      "delay" : 1000,
      "slots" : {
        "failure" : { "command" :  "exit" },
        "success": { "command": "exit" }
      }
    }
  ]
}
```

위 테스트 과정을 통해 서버 연결 해제 과정을 개선하고 DB 처리 시
SQL 서버의 ODBC 드라이버에서 연결 풀링을 해주는 것을 확인했다.

락 처리만 깔끔하면 전체적인 서버 처리 모델과 DB 처리가 안정적이
될 수 있다는 점을 확인했다.

정리

프로젝트 초기에 서버의 중요한 기반을 단위 테스트, 단위 성능 테스트,
부하 테스트를 하면 매우 안정적이고 성능 요건을 만족하는 기반을 확보할 수 있다.

서버 기반은 프로젝트의 매 단계 진행에도 영향을 미치므로 개발 효율도 올라가고
출시 전 부하 테스트 요건을 쉽게 만족할 수 있으므로 출시 전 리스크도 사전에
줄일 수 있다.

또한, 이 모든 과정이 한 두달 내에 완료될 수 있기 때문에 비용 면에서도 효율적이다.