Paxospp

COMS 4995 Design with C++

Jiawei Zhang, Luofei Zhang, Xijiao Li @ 04/08/21

Acknowledgements

- Bjarne Stroustrup
- Leslie Lamport
- Robert Morris

Overview

- Introduction to Paxos
- Paxos Library
    - Client & Server
- Test & Performance
- Example Project
    - KVStore

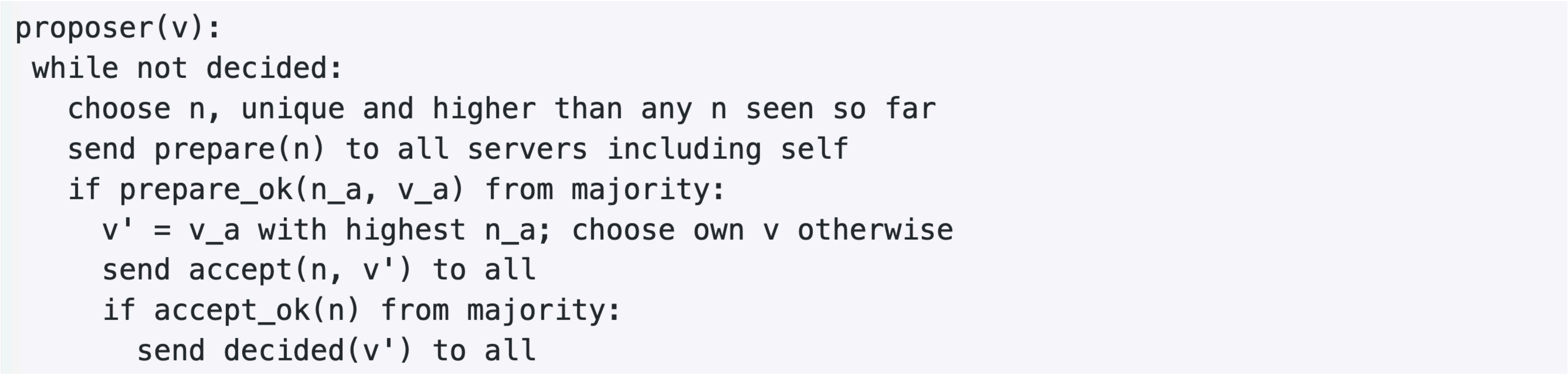
Introduction to Paxos

- Distributed Consensus & Implementations
- Pseudocode of Paxos

- Distributed Consensus
  - Paxos
  - Raft
  - ZooKeeper Atomic Broadcast
  - Proof-of-Work Systems
    - Bitcoin
  - Lockstep Anti-Cheating
    - Age of Empires

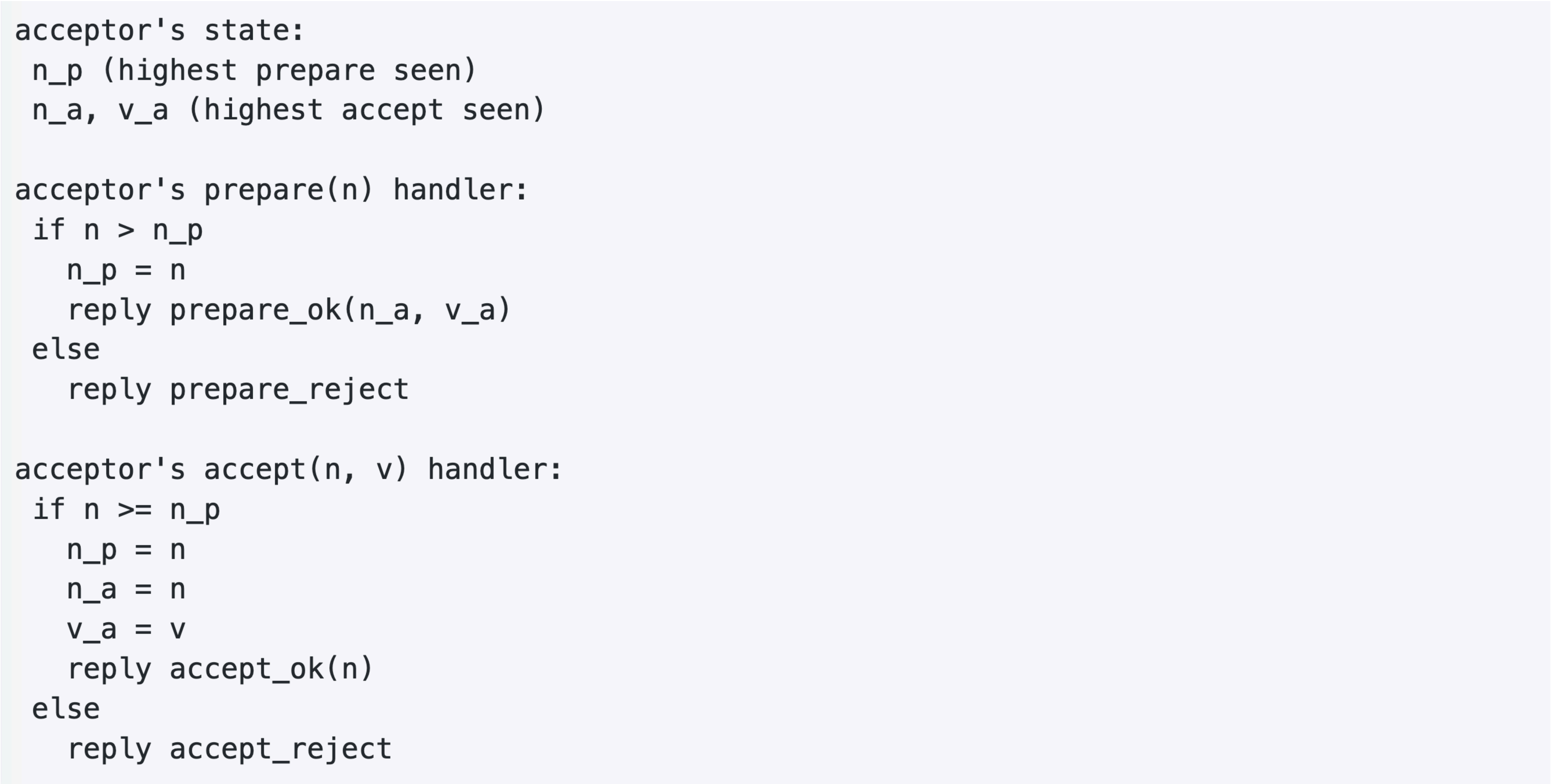
- Implementations
  - Chubby
    - coarse grained lock service
  - etcd
    - a distributed key value store
  - Apache ZooKeeper
    - a centralized service for maintaining configuration information, naming, providing distributed synchronization

- Pseudocode of Paxos
  - Paxos as a client



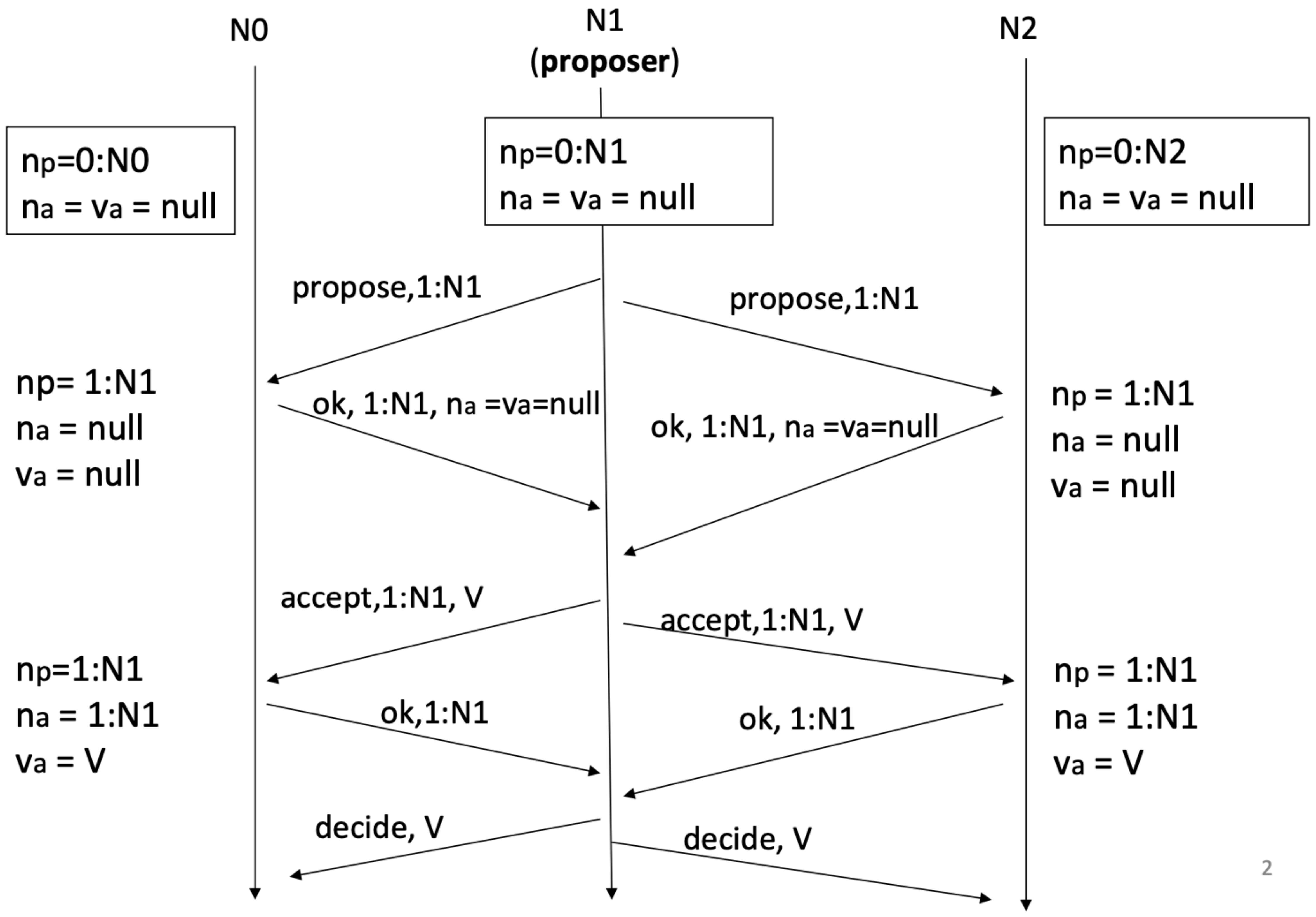
```
proposer(v):
 while not decided:
   choose n, unique and higher than any n seen so far
   send prepare(n) to all servers including self
   if prepare_ok(n_a, v_a) from majority:
     v' = v_a with highest n_a; choose own v otherwise
     send accept(n, v') to all
     if accept_ok(n) from majority:
       send decided(v') to all
```

- Pseudocode of Paxos
  - Paxos as a server



```
acceptor's state:
 n_p (highest prepare seen)
 n_a, v_a (highest accept seen)

acceptor's prepare(n) handler:
 if n > n_p
   n_p = n
   reply prepare_ok(n_a, v_a)
 else
   reply prepare_reject

acceptor's accept(n, v) handler:
 if n >= n_p
   n_p = n
   n_a = n
   v_a = v
   reply accept_ok(n)
 else
   reply accept_reject
```

- Examples scenario
  - Single proposer



N0

N1 (**proposer**)

N2

n_p=0:N0
n_a = v_a = null

n_p=0:N1
n_a = v_a = null

n_p=0:N2
n_a = v_a = null

propose,1:N1

propose,1:N1

n_p= 1:N1
n_a = null
v_a = null

ok, 1:N1, n_a =v_a=null

ok, 1:N1, n_a =v_a=null

n_p = 1:N1
n_a = null
v_a = null

accept,1:N1, V

accept,1:N1, V

n_p=1:N1
n_a = 1:N1
v_a = V

ok,1:N1

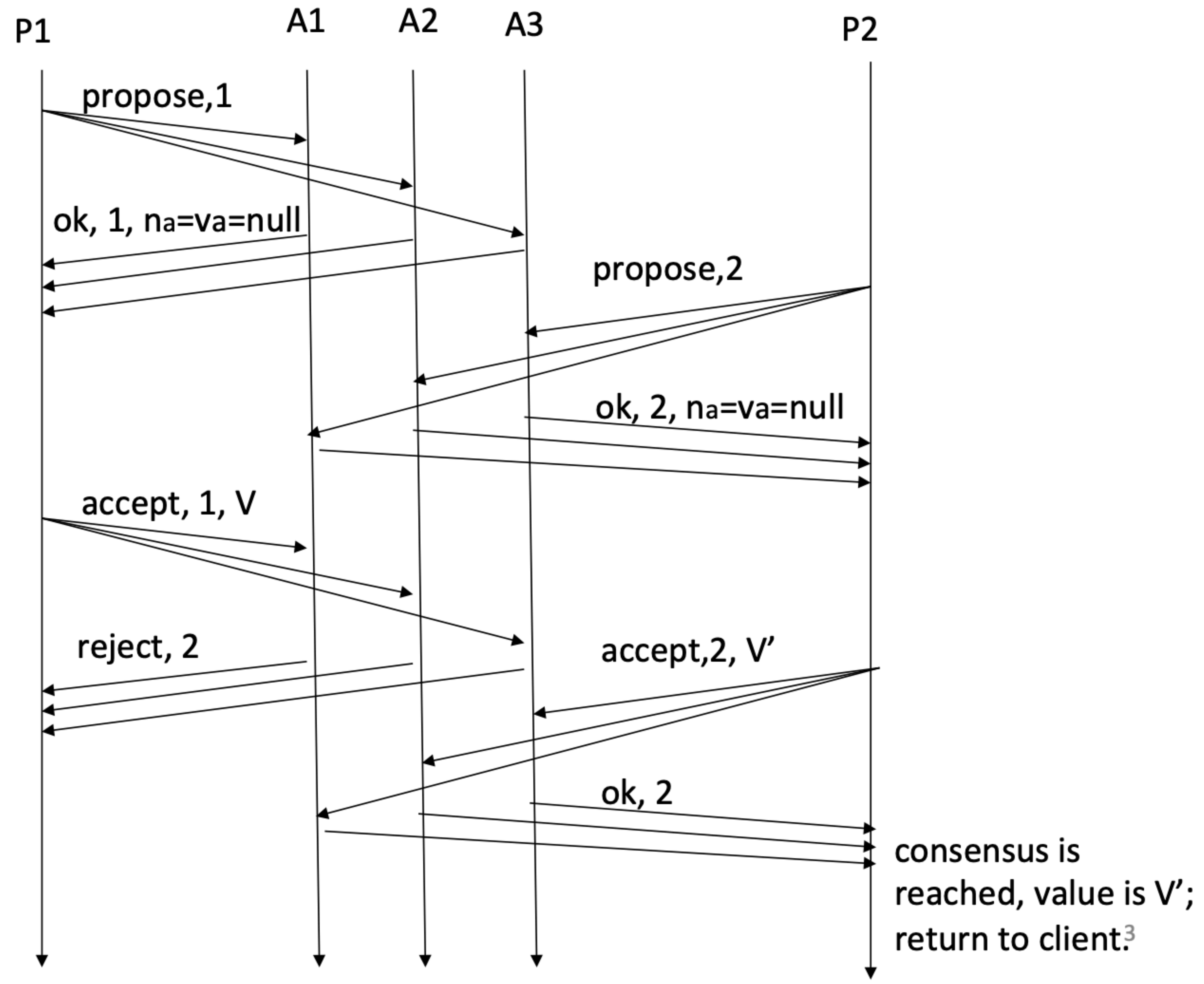
ok, 1:N1

n_p = 1:N1
n_a = 1:N1
v_a = V

decide, V

decide, V

- Examples scenario
    - Concurrent proposers



Paxospp Library

- Paxos::Service API
- PaxosServiceImpl Class & API
- PaxosServiceImpl Function Details

- Paxos::Service API

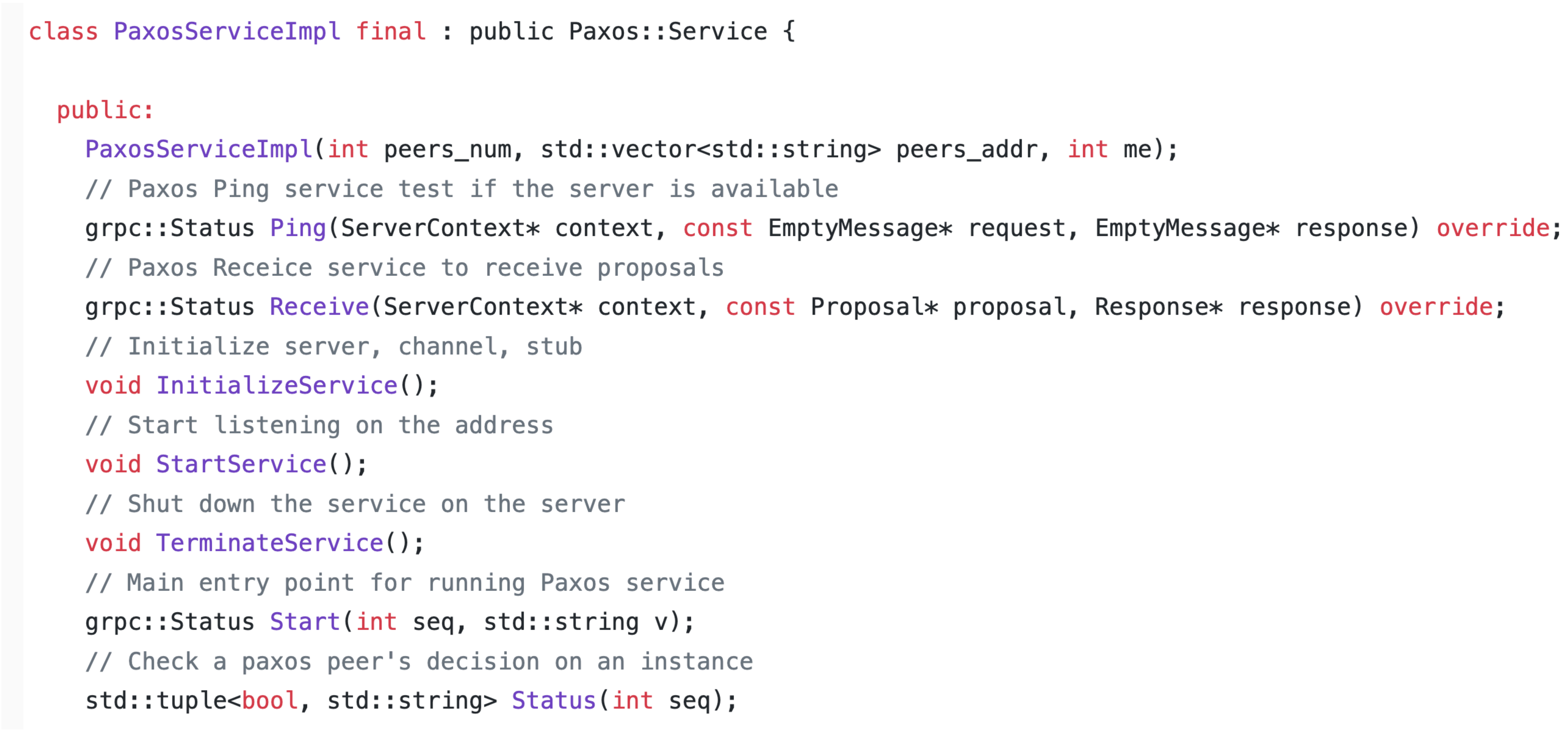
```
service Paxos {
  rpc Ping (EmptyMessage) returns (EmptyMessage) {}
  rpc Receive (Proposal) returns (Response) {}
}
```

```
message Proposal {
  string    type = 1;
  int32     proposed_num = 2;
  int32     seq = 3;
  string    value = 4;
  int32     me = 5;
  int32     done = 6;
}
```

```
message Response {
  string    type = 1;
  bool      approved = 2;
  int32     number = 3;
  string    value = 4;
  int32     me = 5;
  int32     done = 6;
}
```

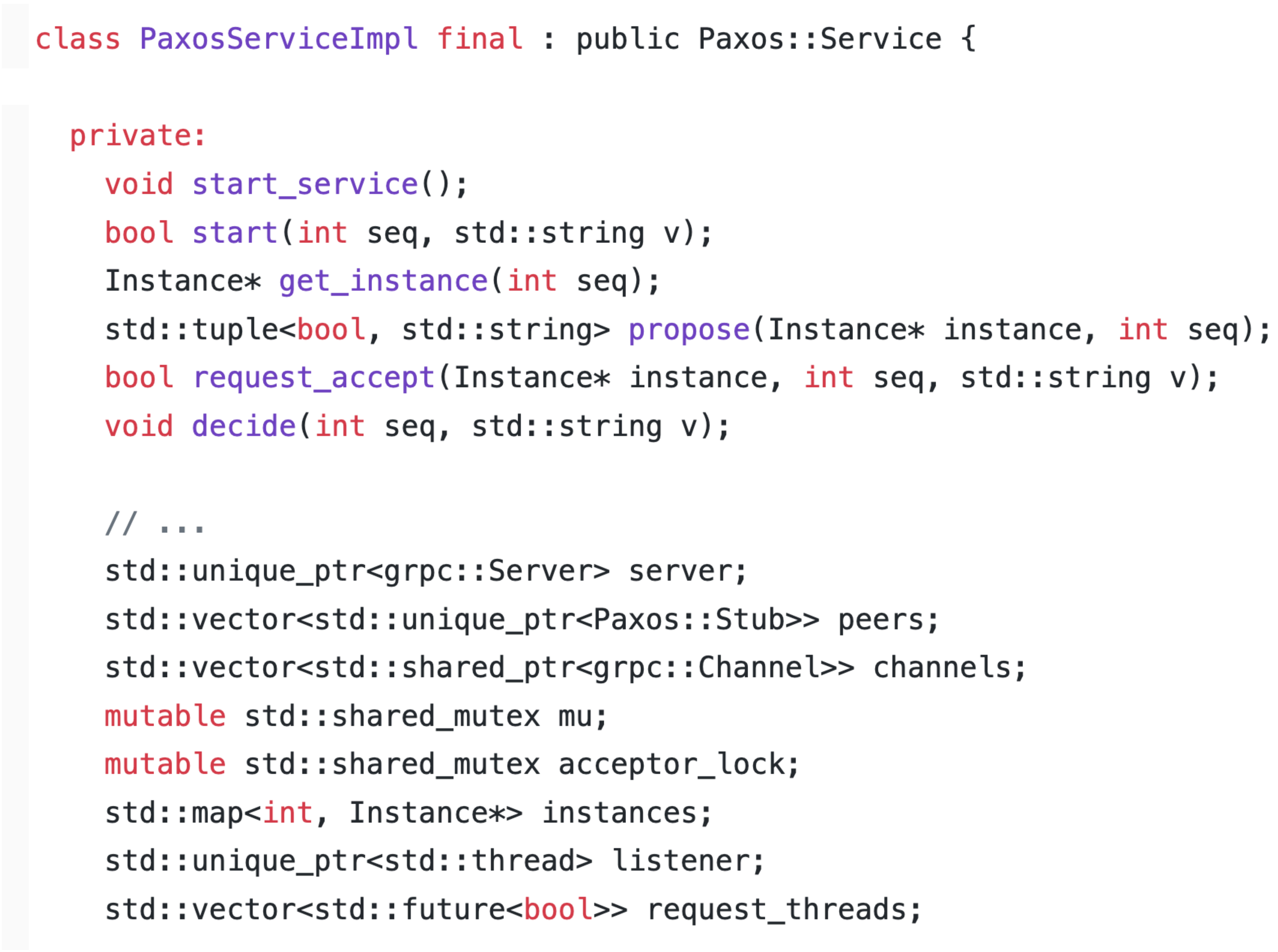
```
message EmptyMessage {}
```

- PaxosServiceImpl Class & API



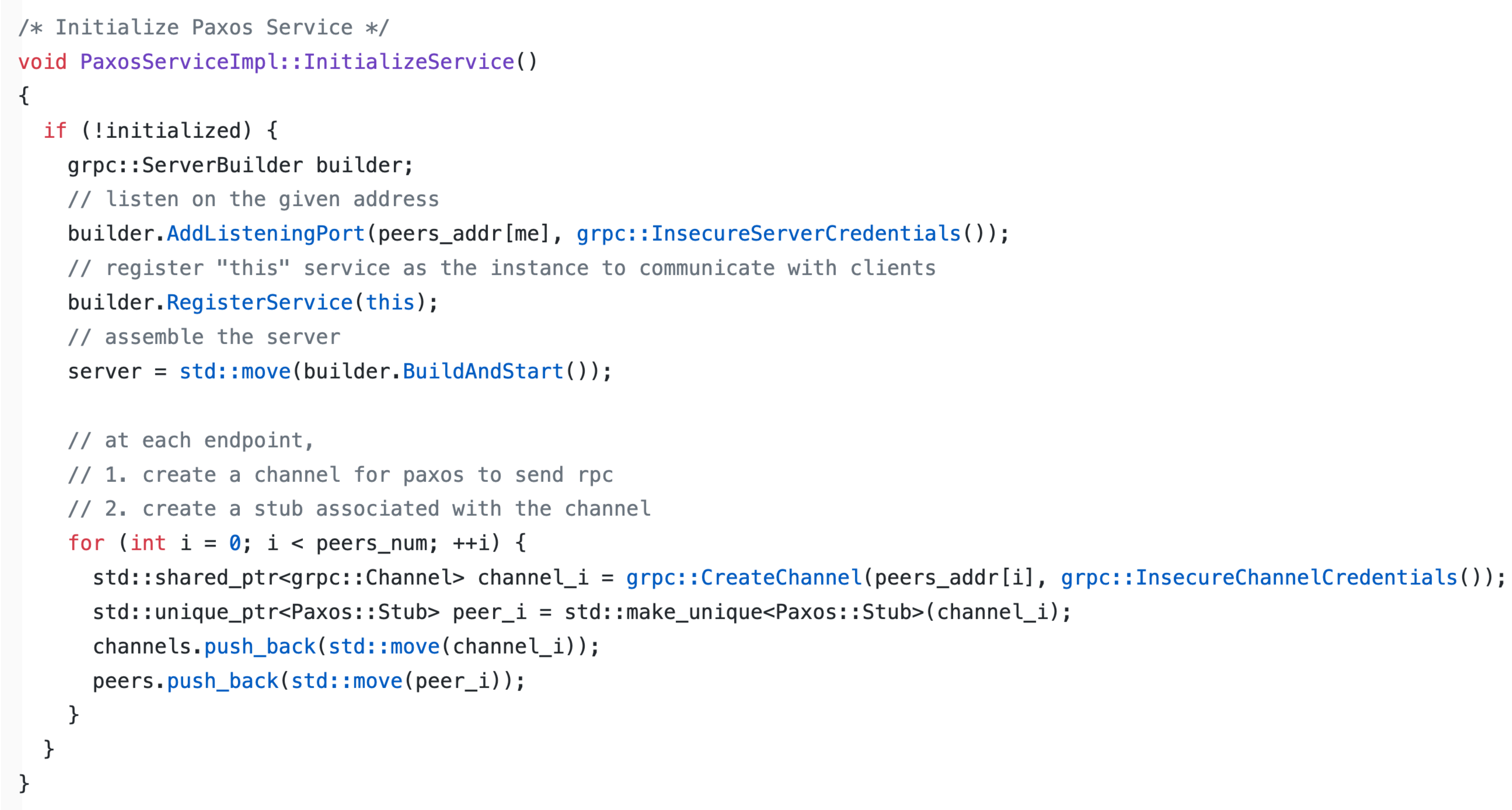
```
class PaxosServiceImpl final : public Paxos::Service {

  public:
    PaxosServiceImpl(int peers_num, std::vector<std::string> peers_addr, int me);
    // Paxos Ping service test if the server is available
    grpc::Status Ping(ServerContext* context, const EmptyMessage* request, EmptyMessage* response) override;
    // Paxos Receice service to receive proposals
    grpc::Status Receive(ServerContext* context, const Proposal* proposal, Response* response) override;
    // Initialize server, channel, stub
    void InitializeService();
    // Start listening on the address
    void StartService();
    // Shut down the service on the server
    void TerminateService();
    // Main entry point for running Paxos service
    grpc::Status Start(int seq, std::string v);
    // Check a paxos peer's decision on an instance
    std::tuple<bool, std::string> Status(int seq);
```

- PaxosServiceImpl Class & API



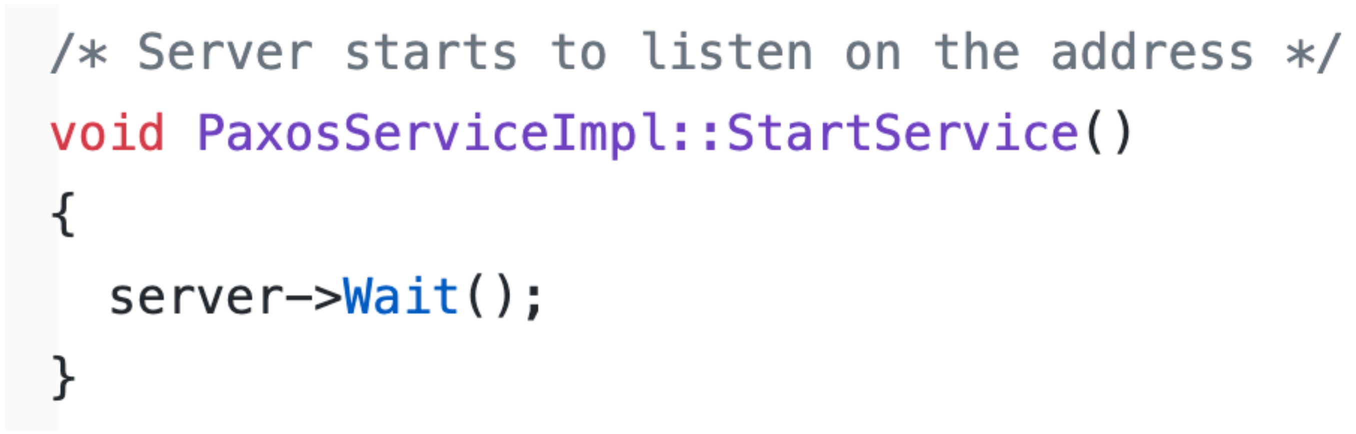
```
class PaxosServiceImpl final : public Paxos::Service {

  private:
    void start_service();
    bool start(int seq, std::string v);
    Instance* get_instance(int seq);
    std::tuple<bool, std::string> propose(Instance* instance, int seq);
    bool request_accept(Instance* instance, int seq, std::string v);
    void decide(int seq, std::string v);

    // ...
    std::unique_ptr<grpc::Server> server;
    std::vector<std::unique_ptr<Paxos::Stub>> peers;
    std::vector<std::shared_ptr<grpc::Channel>> channels;
    mutable std::shared_mutex mu;
    mutable std::shared_mutex acceptor_lock;
    std::map<int, Instance*> instances;
    std::unique_ptr<std::thread> listener;
    std::vector<std::future<bool>> request_threads;
```

- PaxosServiceImpl Function Details



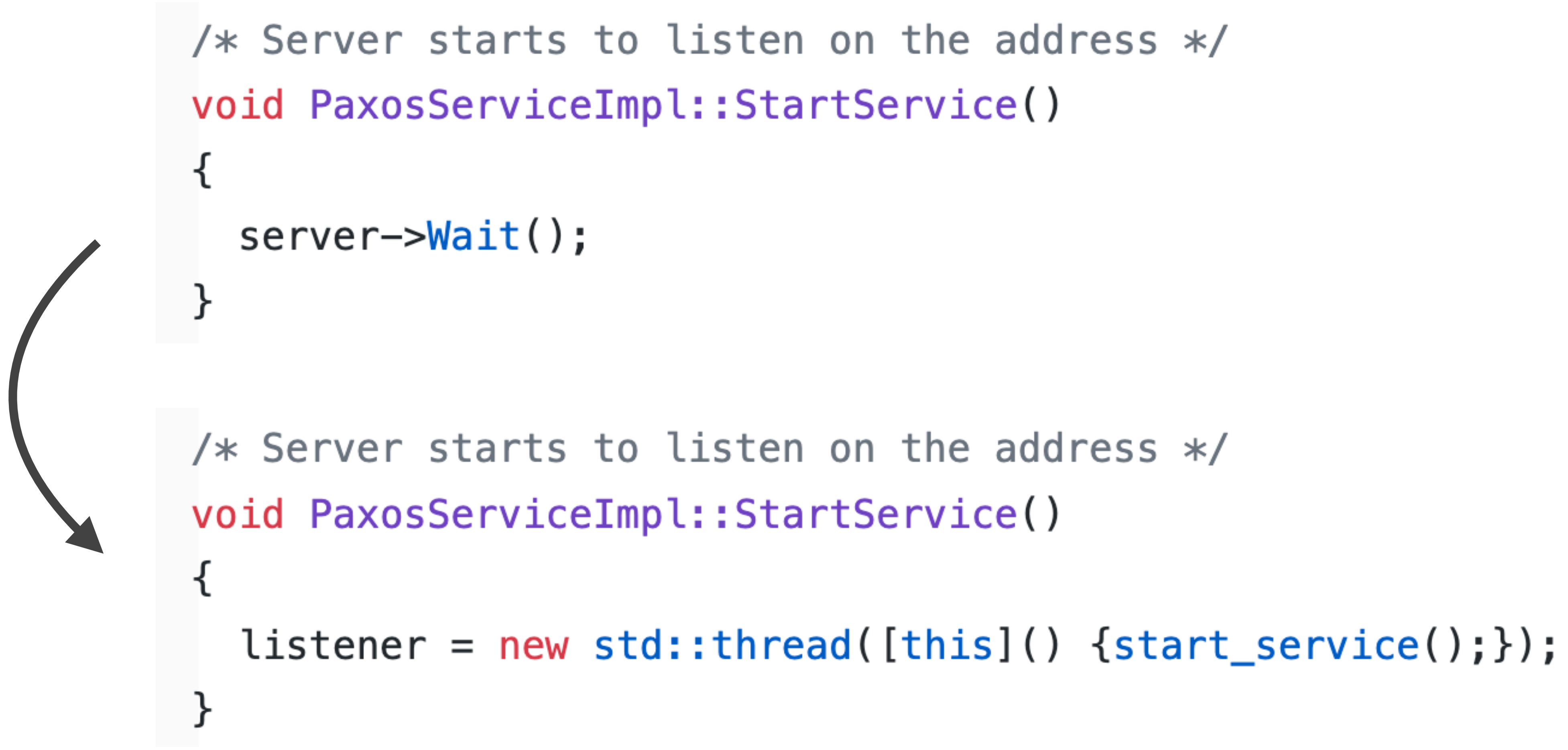
```
/* Initialize Paxos Service */
void PaxosServiceImpl::InitializeService()
{
  if (!initialized) {
    grpc::ServerBuilder builder;
    // listen on the given address
    builder.AddListeningPort(peers_addr[me], grpc::InsecureServerCredentials());
    // register "this" service as the instance to communicate with clients
    builder.RegisterService(this);
    // assemble the server
    server = std::move(builder.BuildAndStart());

    // at each endpoint,
    // 1. create a channel for paxos to send rpc
    // 2. create a stub associated with the channel
    for (int i = 0; i < peers_num; ++i) {
      std::shared_ptr<grpc::Channel> channel_i = grpc::CreateChannel(peers_addr[i], grpc::InsecureChannelCredentials());
      std::unique_ptr<Paxos::Stub> peer_i = std::make_unique<Paxos::Stub>(channel_i);
      channels.push_back(std::move(channel_i));
      peers.push_back(std::move(peer_i));
    }
  }
}
```

- Problem #1
    - Listener will keep blocking



```
/* Server starts to listen on the address */
void PaxosServiceImpl::StartService()
{
  server->Wait();
}
```

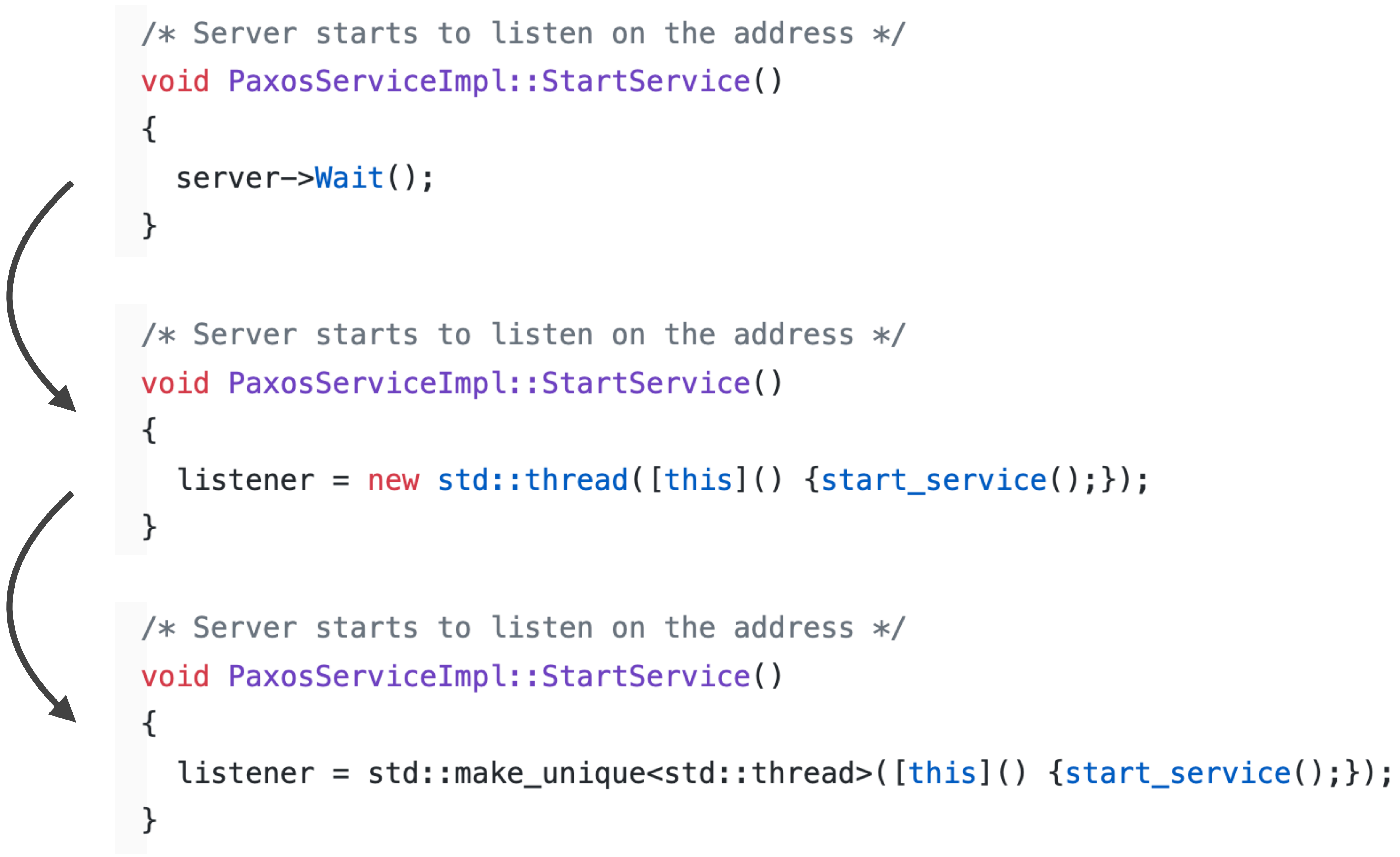
- Improvement #1
  - Listener will keep blocking



```
/* Server starts to listen on the address */
void PaxosServiceImpl::StartService()
{
  server->Wait();
}
```

```
/* Server starts to listen on the address */
void PaxosServiceImpl::StartService()
{
  listener = new std::thread([this]() {start_service();});
}
```

- Improvement #2
  - Change the type of `listener` from thread* to unique_ptr<thread>

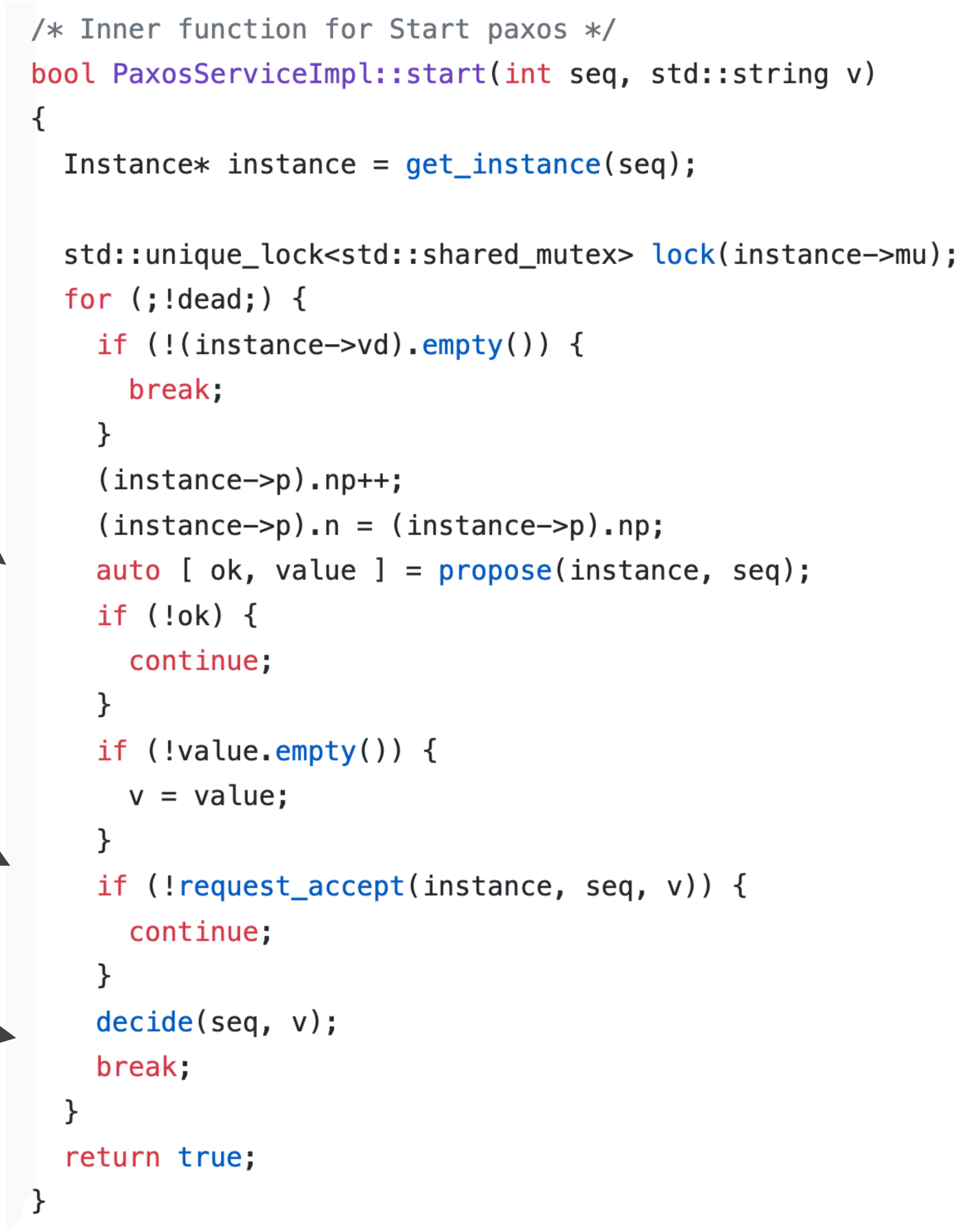


```
/* Server starts to listen on the address */
void PaxosServiceImpl::StartService()
{
  server->Wait();
}
```

```
/* Server starts to listen on the address */
void PaxosServiceImpl::StartService()
{
  listener = new std::thread([this]() {start_service();});
}
```

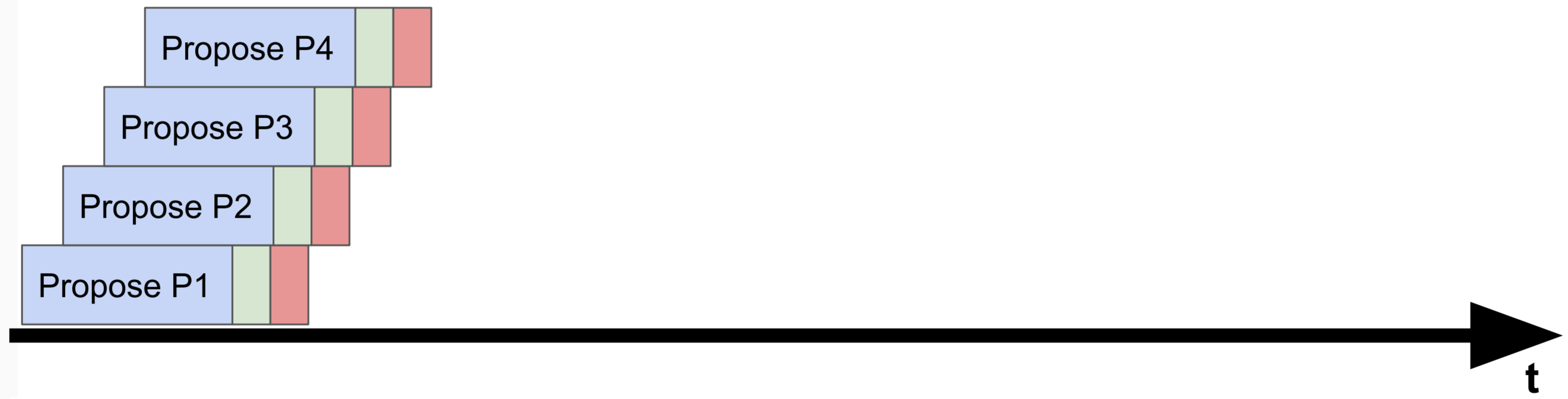
```
/* Server starts to listen on the address */
void PaxosServiceImpl::StartService()
{
  listener = std::make_unique<std::thread>([this]() {start_service();});
}
```

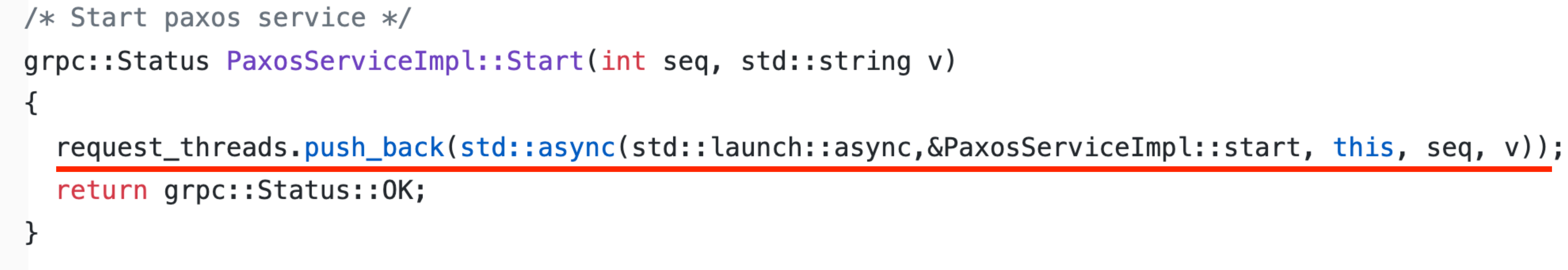
- PaxosServiceImpl Function Details
  - Propose stage
    - Send proposal to all peers
    - Count responses
    - Learn previous consensus (if any)
  - Accept stage
    - Send proposal to all peers
    - Count responses
    - Decide stage



```cpp
/* Inner function for Start paxos */
bool PaxosServiceImpl::start(int seq, std::string v)
{
  Instance* instance = get_instance(seq);

  std::unique_lock<std::shared_mutex> lock(instance->mu);
  for (;!dead;) {
    if (!(instance->vd).empty()) {
      break;
    }
    (instance->p).np++;
    (instance->p).n = (instance->p).np;
    auto [ ok, value ] = propose(instance, seq);
    if (!ok) {
      continue;
    }
    if (!value.empty()) {
      v = value;
    }
    if (!request_accept(instance, seq, v)) {
      continue;
    }
    decide(seq, v);
    break;
  }
  return true;
}
```

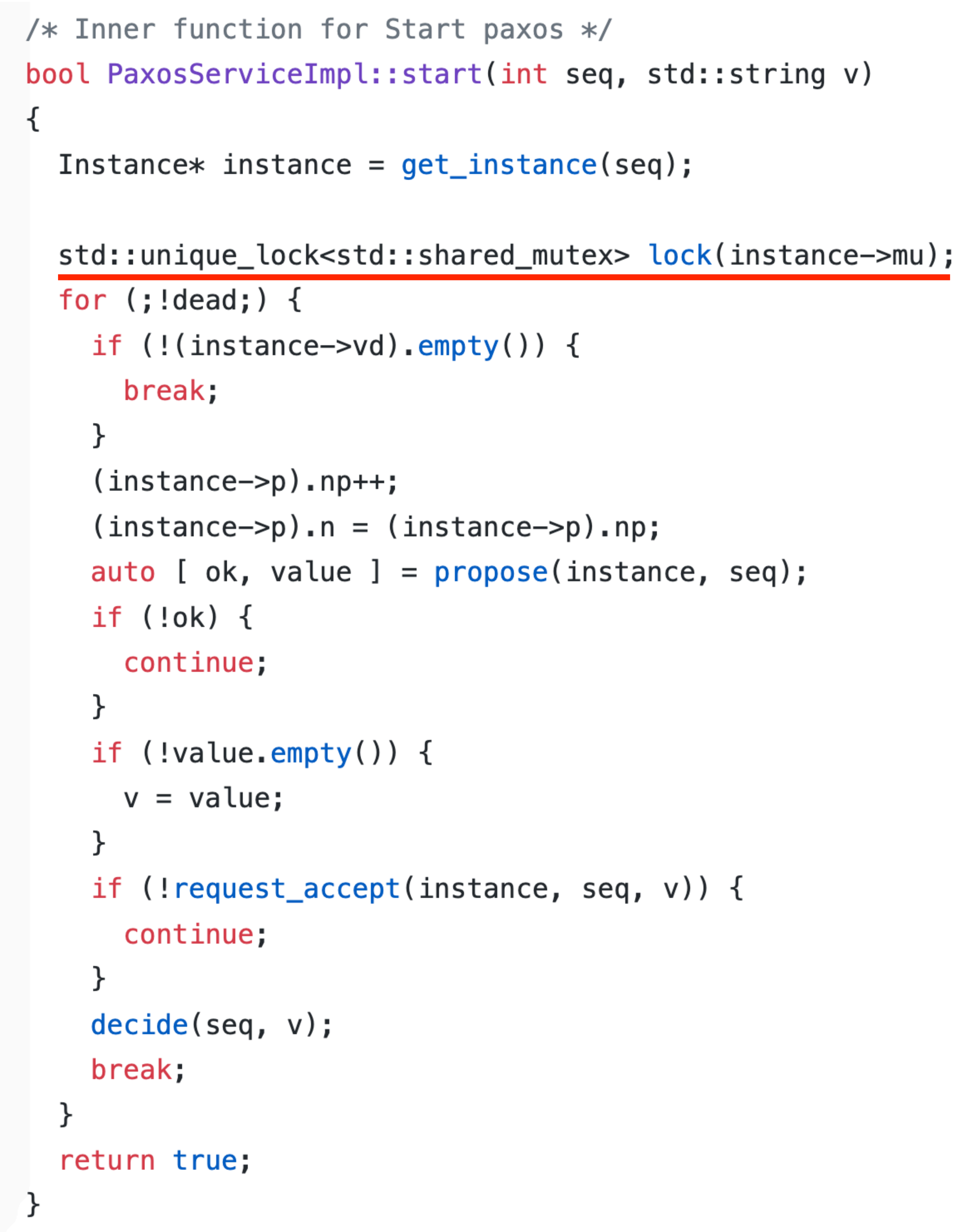
- Improvement #1
  - Handle parallel requests





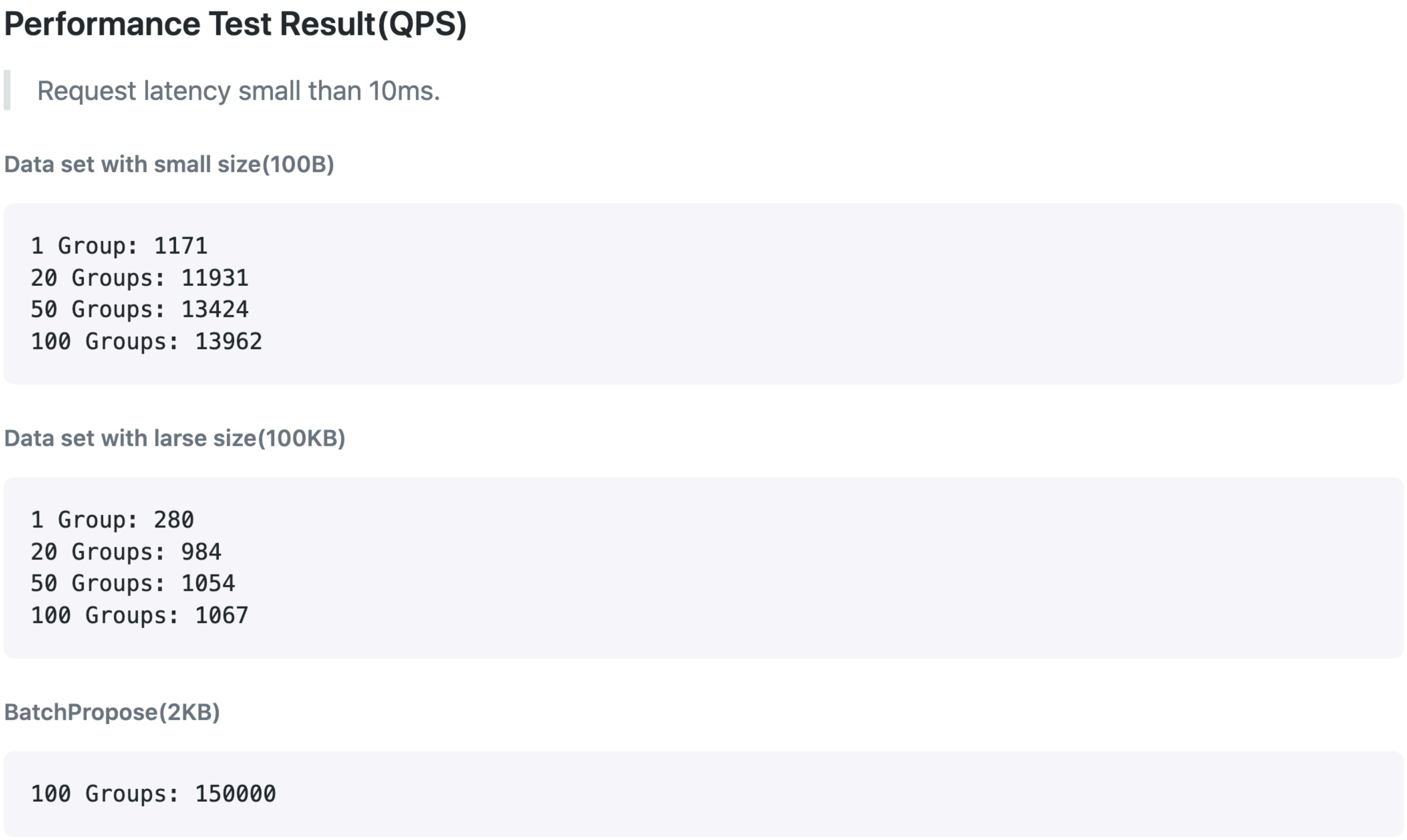
```
/* Start paxos service */
grpc::Status PaxosServiceImpl::Start(int seq, std::string v)
{
  request_threads.push_back(std::async(std::launch::async,&PaxosServiceImpl::start, this, seq, v));
  return grpc::Status::OK;
}
```

- Improvement #2
    - Use read/write lock to ensure only one thread is updating an instance at any given time.



```
/* Inner function for Start paxos */
bool PaxosServiceImpl::start(int seq, std::string v)
{
  Instance* instance = get_instance(seq);

  std::unique_lock<std::shared_mutex> lock(instance->mu);
  for (;!dead;) {
    if (!(instance->vd).empty()) {
      break;
    }
    (instance->p).np++;
    (instance->p).n = (instance->p).np;
    auto [ ok, value ] = propose(instance, seq);
    if (!ok) {
      continue;
    }
    if (!value.empty()) {
      v = value;
    }
    if (!request_accept(instance, seq, v)) {
      continue;
    }
    decide(seq, v);
    break;
  }
  return true;
}
```

Test & Performance



Performance Test Result(QPS)

> Request latency small than 10ms.

Data set with small size(100B)

```
1 Group: 1171
20 Groups: 11931
50 Groups: 13424
100 Groups: 13962
```

Data set with larse size(100KB)

```
1 Group: 280
20 Groups: 984
50 Groups: 1054
100 Groups: 1067
```

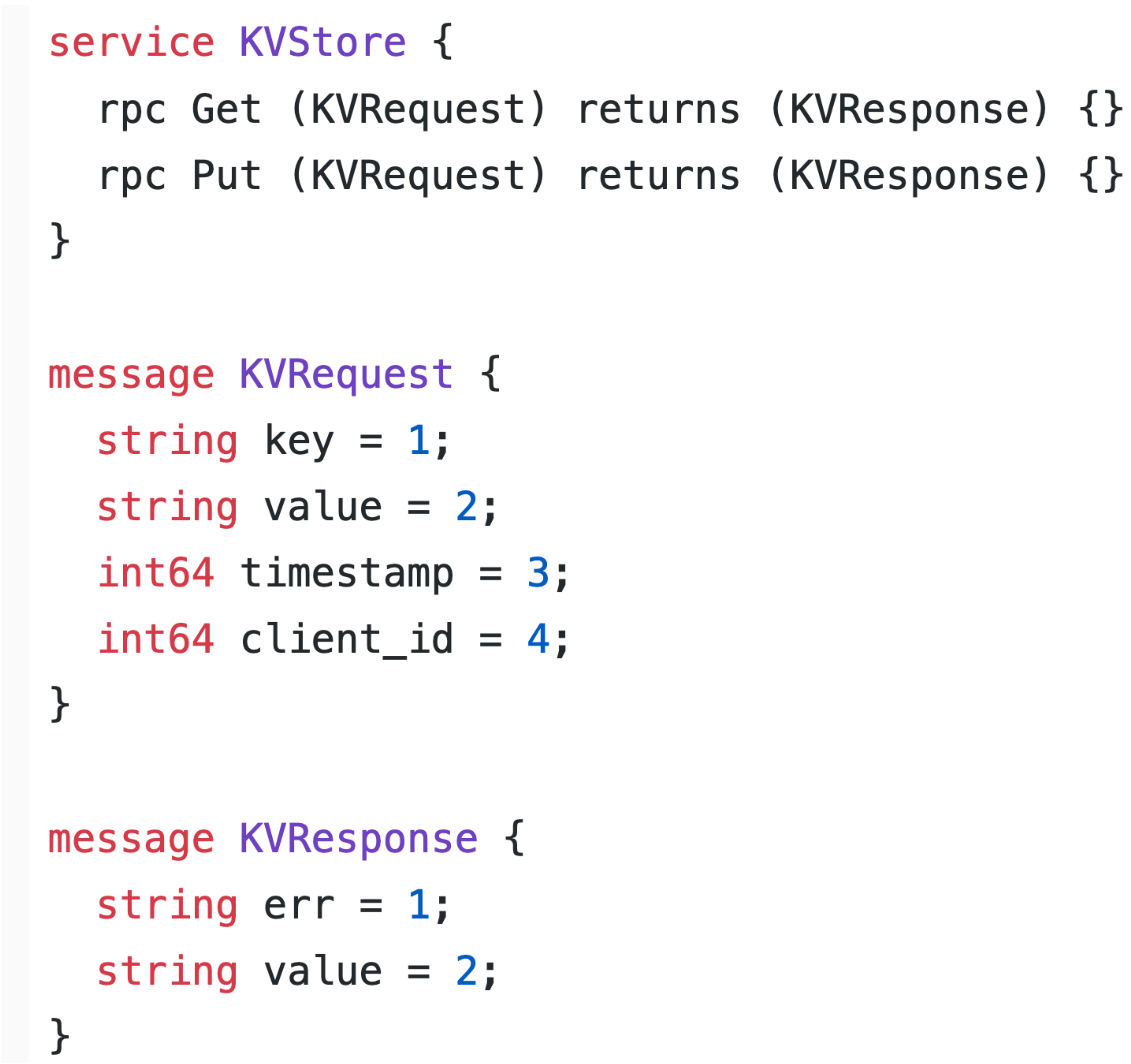
BatchPropose(2KB)

```
100 Groups: 150000
```

Example Project (v1.2)

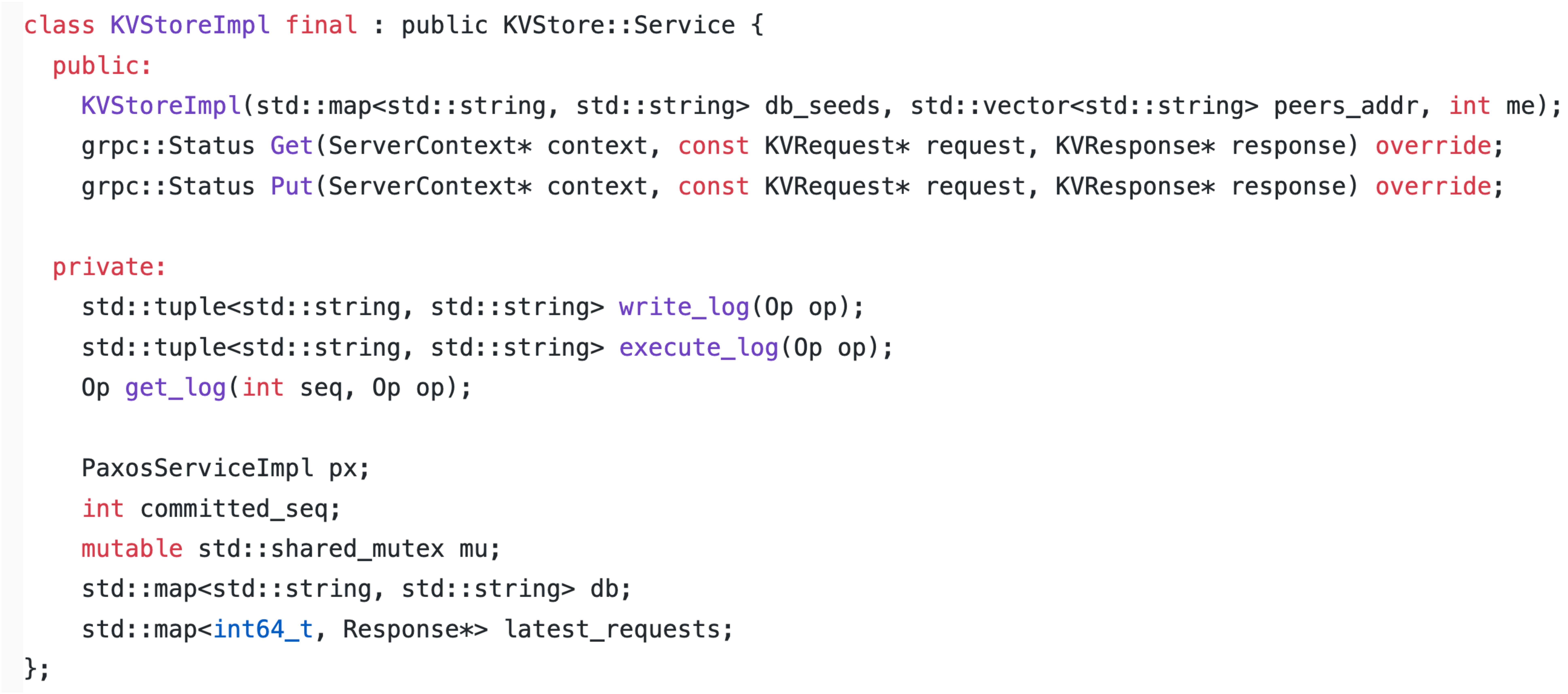
- KVStore service
  - A simple KVStore lib based on our implementation of Paxospp
  - Also a good illustration of how to wrap your own code around Paxospp to help your services in synchronizing
  - Can be applied to synchronize the state from a single node to the other nodes to form a multi-copy cluster and handling fail-over automatically

- KVStore::Service API



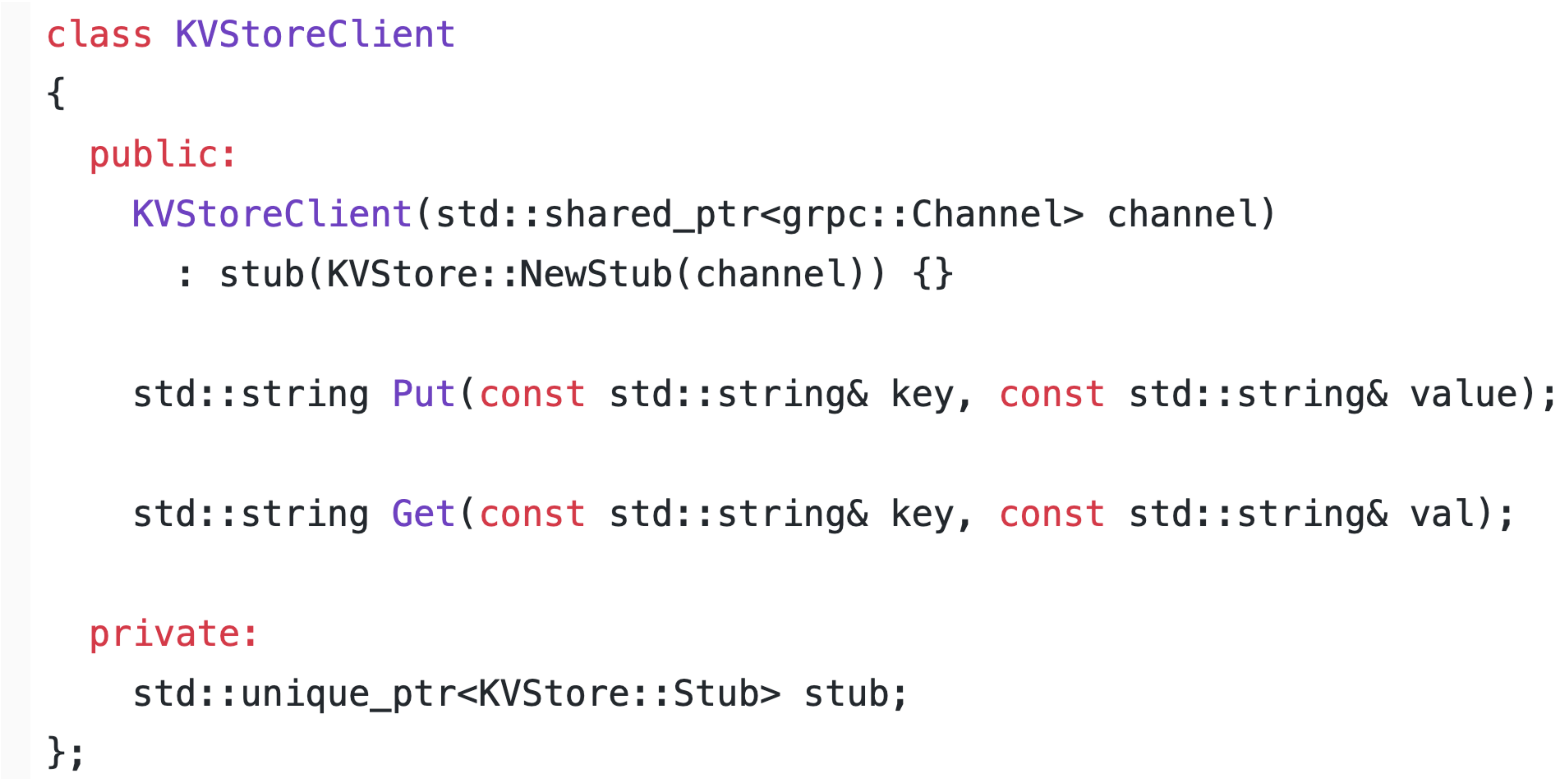
```
service KVStore {
  rpc Get (KVRequest) returns (KVResponse) {}
  rpc Put (KVRequest) returns (KVResponse) {}
}

message KVRequest {
  string key = 1;
  string value = 2;
  int64 timestamp = 3;
  int64 client_id = 4;
}

message KVResponse {
  string err = 1;
  string value = 2;
}
```

- KVStoreImpl Class & API



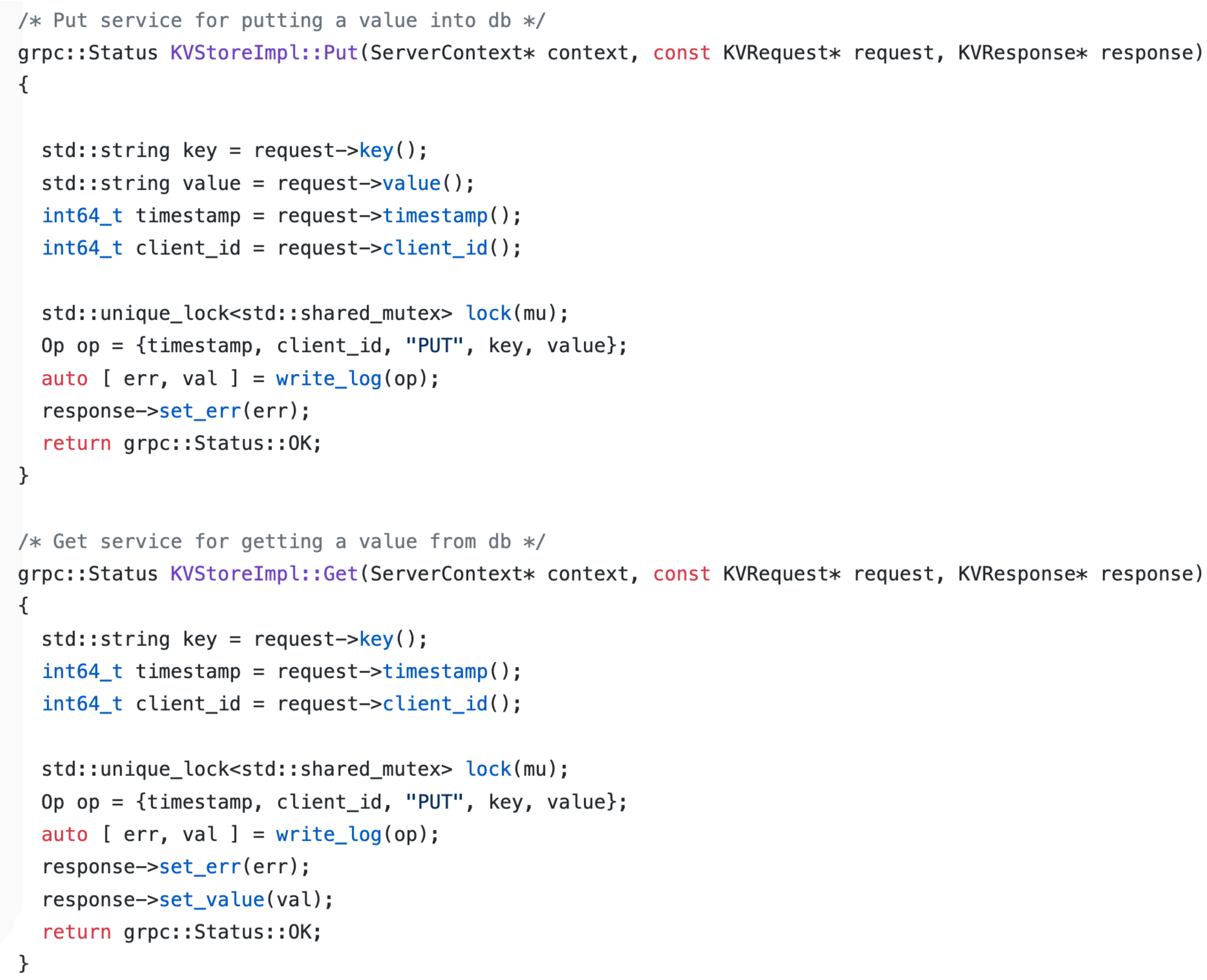
```
class KVStoreImpl final : public KVStore::Service {
  public:
    KVStoreImpl(std::map<std::string, std::string> db_seeds, std::vector<std::string> peers_addr, int me);
    grpc::Status Get(ServerContext* context, const KVRequest* request, KVResponse* response) override;
    grpc::Status Put(ServerContext* context, const KVRequest* request, KVResponse* response) override;

  private:
    std::tuple<std::string, std::string> write_log(Op op);
    std::tuple<std::string, std::string> execute_log(Op op);
    Op get_log(int seq, Op op);

    PaxosServiceImpl px;
    int committed_seq;
    mutable std::shared_mutex mu;
    std::map<std::string, std::string> db;
    std::map<int64_t, Response*> latest_requests;
};
```

- KVStoreClient Class & API



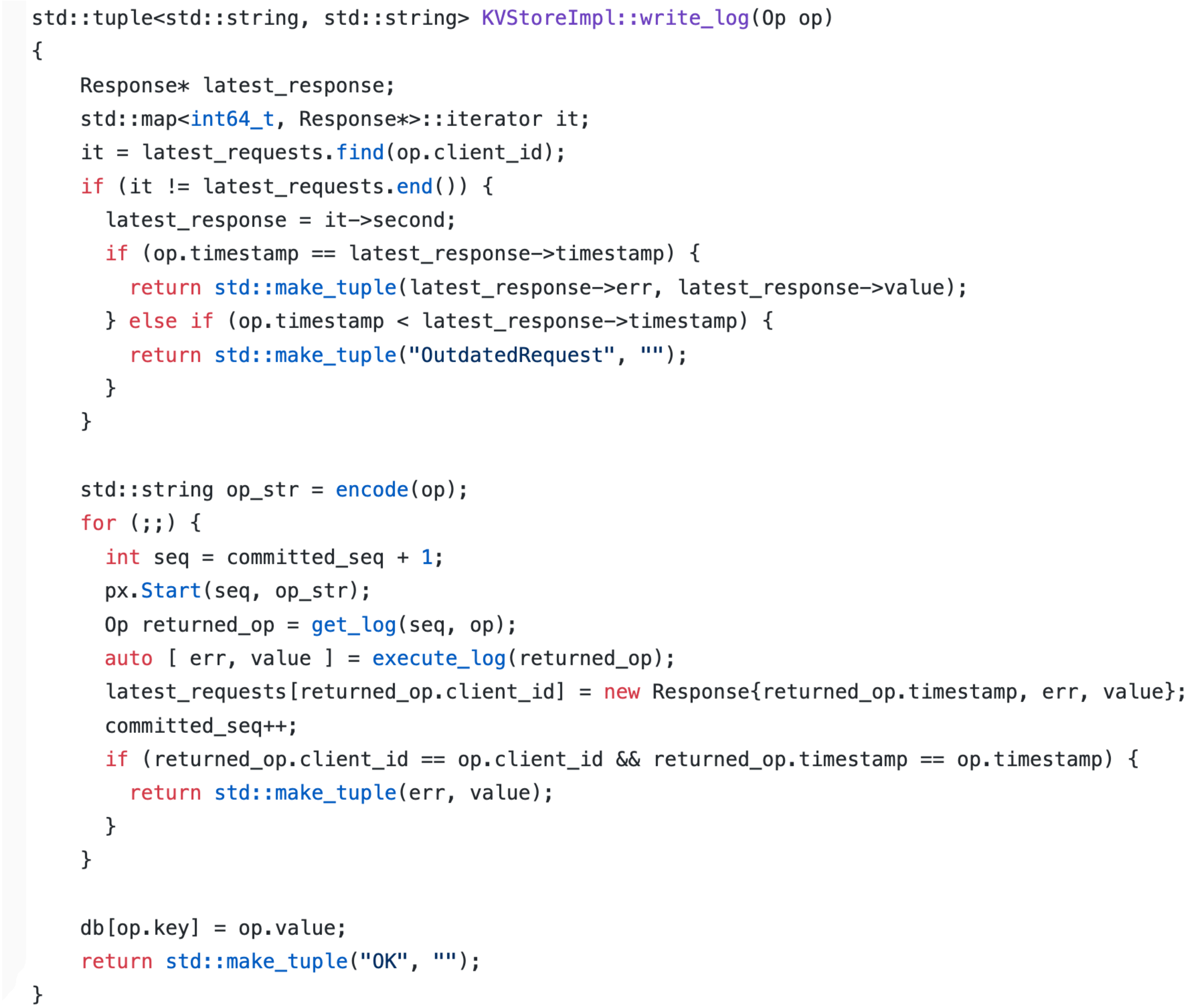
```
class KVStoreClient
{
  public:
    KVStoreClient(std::shared_ptr<grpc::Channel> channel)
      : stub(KVStore::NewStub(channel)) {}

    std::string Put(const std::string& key, const std::string& value);

    std::string Get(const std::string& key, const std::string& val);

  private:
    std::unique_ptr<KVStore::Stub> stub;
};
```

- KVStoreImpl Function Details



```
/* Put service for putting a value into db */
grpc::Status KVStoreImpl::Put(ServerContext* context, const KVRequest* request, KVResponse* response)
{

  std::string key = request->key();
  std::string value = request->value();
  int64_t timestamp = request->timestamp();
  int64_t client_id = request->client_id();

  std::unique_lock<std::shared_mutex> lock(mu);
  Op op = {timestamp, client_id, "PUT", key, value};
  auto [ err, val ] = write_log(op);
  response->set_err(err);
  return grpc::Status::OK;
}

/* Get service for getting a value from db */
grpc::Status KVStoreImpl::Get(ServerContext* context, const KVRequest* request, KVResponse* response)
{
  std::string key = request->key();
  int64_t timestamp = request->timestamp();
  int64_t client_id = request->client_id();

  std::unique_lock<std::shared_mutex> lock(mu);
  Op op = {timestamp, client_id, "PUT", key, value};
  auto [ err, val ] = write_log(op);
  response->set_err(err);
  response->set_value(val);
  return grpc::Status::OK;
}
```

- KVStoreImpl Function Details



```
std::tuple<std::string, std::string> KVStoreImpl::write_log(Op op)
{
    Response* latest_response;
    std::map<int64_t, Response*>::iterator it;
    it = latest_requests.find(op.client_id);
    if (it != latest_requests.end()) {
      latest_response = it->second;
      if (op.timestamp == latest_response->timestamp) {
        return std::make_tuple(latest_response->err, latest_response->value);
      } else if (op.timestamp < latest_response->timestamp) {
        return std::make_tuple("OutdatedRequest", "");
      }
    }

    std::string op_str = encode(op);
    for (;;) {
      int seq = committed_seq + 1;
      px.Start(seq, op_str);
      Op returned_op = get_log(seq, op);
      auto [ err, value ] = execute_log(returned_op);
      latest_requests[returned_op.client_id] = new Response{returned_op.timestamp, err, value};
      committed_seq++;
      if (returned_op.client_id == op.client_id && returned_op.timestamp == op.timestamp) {
        return std::make_tuple(err, value);
      }
    }

    db[op.key] = op.value;
    return std::make_tuple("OK", "");
}
```