Paxospp

COMS 4995 Design with C++

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

Acknowledgements

- Bjarne Stroustrup. Modern C++ Features
- Leslie Lamport. Paxos Made Simple Paper
- MIT. 6.824: Distributed Systems
- Columbia University. COMS W4113 Distributed Systems

Overview

- Motivation
- Introduction to Paxos
- Paxos Library
  - Communication: gRPC
  - Proposer & Acceptor
- Test & Performance
- Example Project
  - KVStore & Demo

Motivation

- Why Distributed Systems?
  - Scalibility
  - Fault tolerance

- 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

Motivation

- Why Paxospp?
  - A user friendly cpp library that implement Paxos algorithm
  - Maintain consistency among servers in the cluster
  - Asynchronous request handling – higher throughput and shorter response time

Introduction to Paxos

- Paxos is a consensus algorithm that make a cluster of nodes agree on the same value. It guarantees the system are available to clients as long as the majority of the nodes stay alive, it provides high availability and consistency at the same time.
- Three stages
  - Propose, Accept, and Decide.
- Two roles
  - proposer, acceptor

- **Pseudocode of Paxos**
  - Paxos as proposer

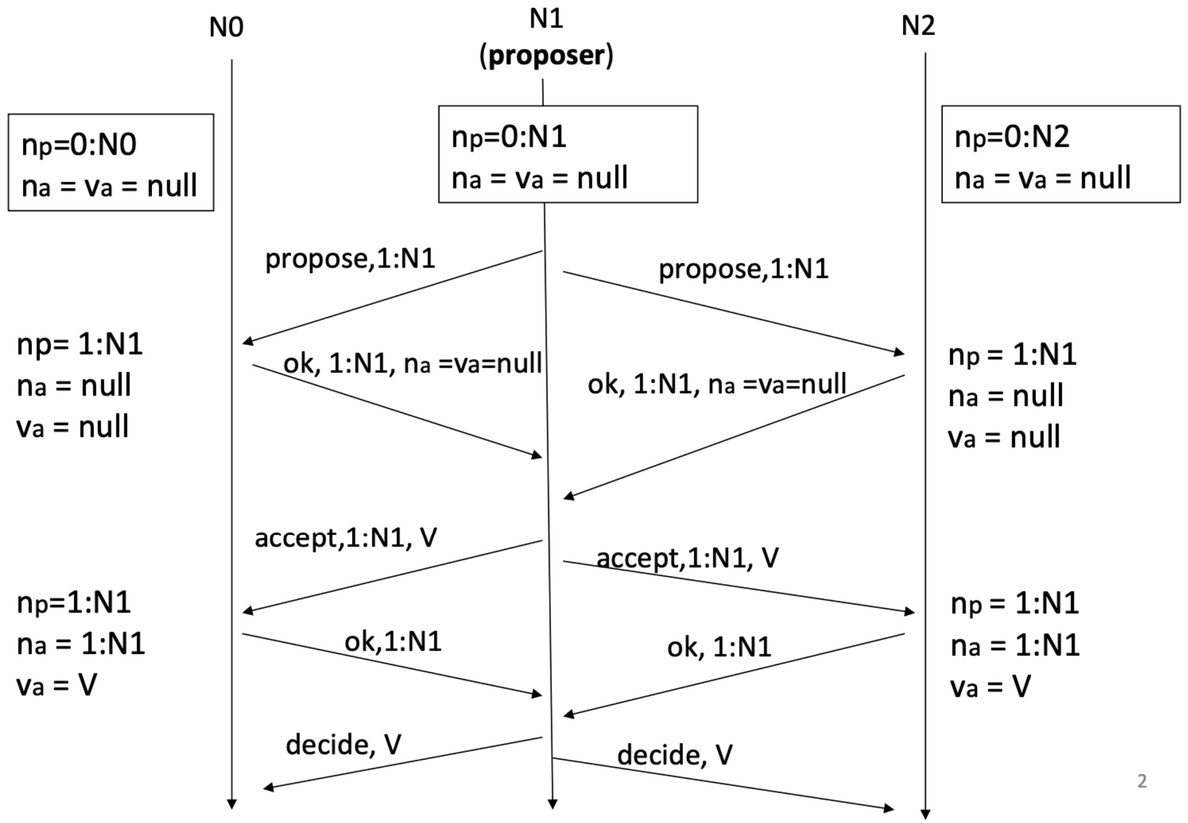
```
proposer(v):
 while not decided:
   choose n, unique and higher than any n seen so far
   send prepare(n) to all servers including self
   if prepare_ok(n_a, v_a) from majority:
     v' = v_a with highest n_a; choose own v otherwise
     send accept(n, v') to all
     if accept_ok(n) from majority: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

- **Pseudocode of Paxos**
  - Paxos as acceptor

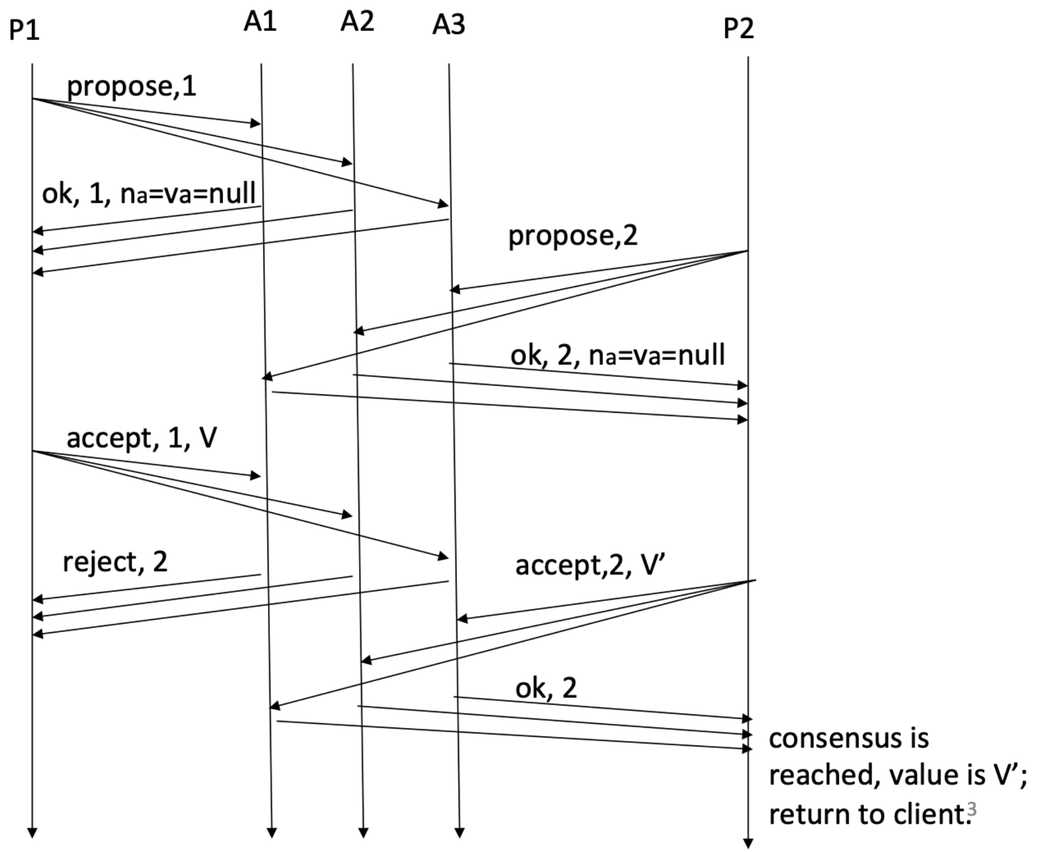
```
acceptor's state:
  n_p (highest prepare seen)
  n_a, v_a (highest accept seen)

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



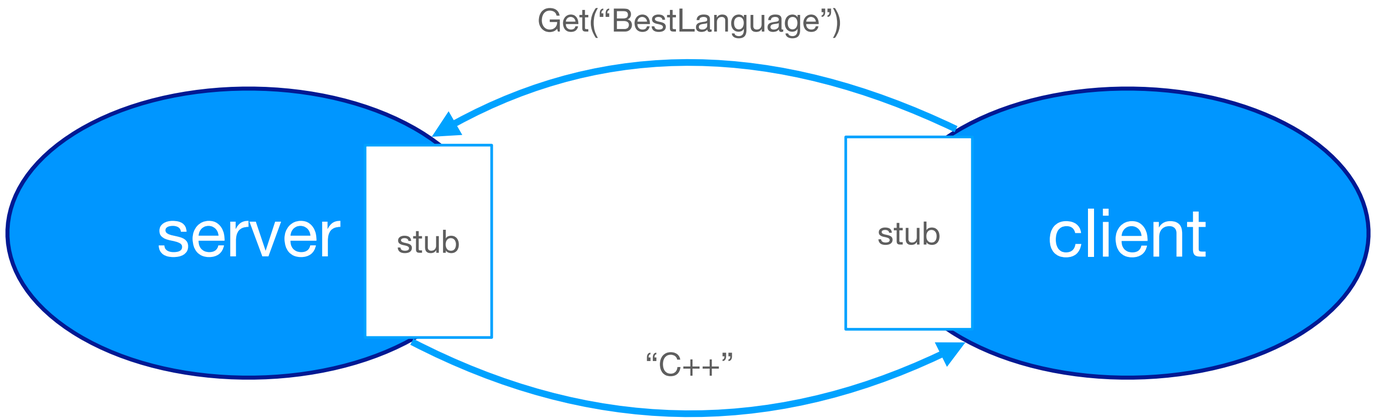
- Examples scenario
  - Concurrent proposers



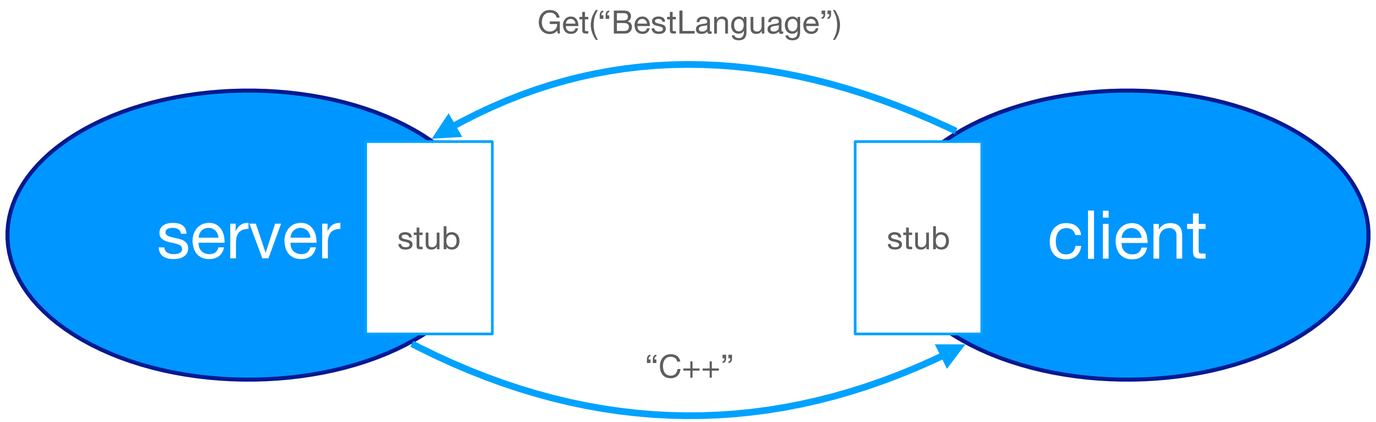
Paxospp Library

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

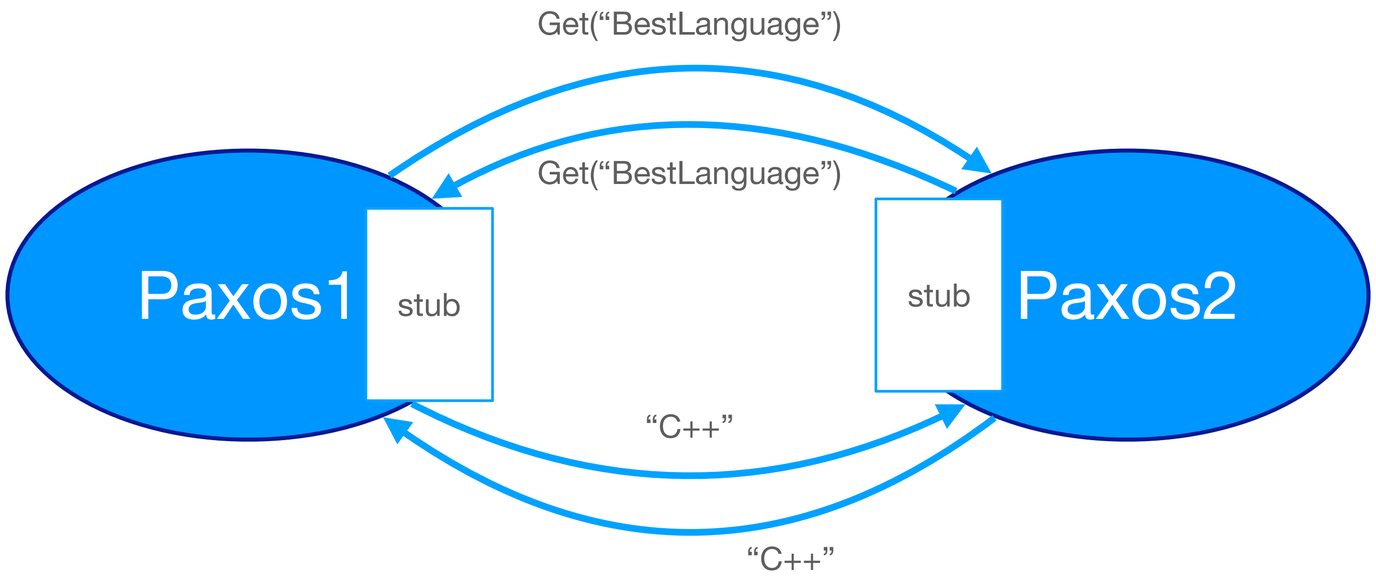
- Paxos::Service API
  - Our implementation makes use of Google's GRPC.
  - Server declares the callable procedures, "stubs" are created.
    - These are methods with identical arguments.
    - There is one each for client & server Client program links to the client stub. Client code can call `Get("...")`, and the stub starts by looking up server's network address ("binding").



- Paxos::Service API
  - If the server is running, the client makes a connection.
  - Now the caller’s arguments are serialized into a message, and then it is sent on the connection to the server. The server stub deserializes them. When the called procedure is done, a reply is sent back to the caller.



- Paxos::Service API
  - If the server is running, the client makes a connection.
  - Now the caller’s arguments are serialized into a message, and then it is sent on the connection to the server. The server stub deserializes them. When the called procedure is done, a reply is sent back to the caller.



- 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
  - Server listening on port will keep blocking

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

- Improvement #1
  - Have a `listener` 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();});
}
```

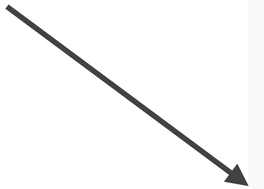
- Improvement #2
  - Change the type of `listener` from thread* to unique_ptr<thread>

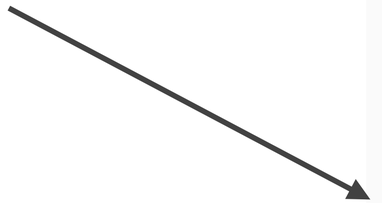
```
/* Server starts to listen on the address */
void PaxosServiceImpl::StartService()
{
  server->Wait();
}
```

```
/* 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





```
/* 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

- Test - NAgreement
  - Data generated on the fly

```
// Nput
for (int i = 0; i < put_size; i++){
  int server_num = random()%peers_num;
  pxs.at(server_num)->Start(i, std::to_string(random()));
}
wait_check(0, put_size-1, pxs, peers_num);
```

```
seq 980: All nodes agree on 201195
seq 981: All nodes agree on 77850
seq 982: All nodes agree on 62490
seq 983: All nodes agree on 283778
seq 984: All nodes agree on 284145
seq 985: All nodes agree on 362964
seq 986: All nodes agree on 407753
seq 987: All nodes agree on 344113
seq 988: All nodes agree on 29650
seq 989: All nodes agree on 348123
seq 990: All nodes agree on 386368
seq 991: All nodes agree on 43978
seq 992: All nodes agree on 343486
seq 993: All nodes agree on 74740
seq 994: All nodes agree on 409994
seq 995: All nodes agree on 49220
seq 996: All nodes agree on 119769
seq 997: All nodes agree on 442277
seq 998: All nodes agree on 89207
seq 999: All nodes agree on 392937
Server shutdown...
Server shutdown...
Server shutdown...
---------------Test_heavy_put: Passed
Luofeis-MBP:build zhangluofei$
```

- Wait and check helper
  - Wait for all the nodes have decided for this seq
  - Only need majority approval, because others can catch up later when they are alive

```
void wait_check(int start, int end, const std::vector<std::unique_ptr<PaxosServiceImpl>>& pxs, int wanted){
  int peers_num = pxs.size();

  for (int i = start; i <= end; i++){
    int ndecided = 0;
    while(ndecided < wanted){
      ndecided = 0;
      for (int j = 0; j < peers_num; j++){
        auto [ decided, _ ] = pxs.at(j)->Status(i);
        if (decided == true){
          ndecided += 1;
        }
      }
    }
```

- Wait and check helper
  - Wait for all the nodes have decided for this seq
  - Only need majority approval, because others can catch up later when they are alive

```
    auto [ decided_0, val_0 ] = pxs.at(0)->Status(i);
    int nagreed = 1;
    for (int j = 1; j < peers_num; j++){
        auto [ decided_1, val_1 ] = pxs.at(j)->Status(i);
        if ( val_0 == val_1 ){
          nagreed++;
        }
    }
    assert( nagreed >= wanted);
    std::cout << "seq " << i << ": " << "All nodes agree on " << val_0 << std::endl;
  }
  return;
}
```

- Test - Unreliable

```
// make a minority of server die
std::set<int> deadServer = {3,4};
for (auto i : deadServer) {
  pxs.at(i)->TerminateService();
}
std::cout << "server 3 and server 4 is down" << std

pxs.at(0)->Start(5,std::to_string(random()));
pxs.at(1)->Start(6,std::to_string(random()));

wait_check(5,6,pxs,3);
```

```
Luofeis-MBP:build zhangluofei$ ./test
0 ok
1 ok
2 ok
3 ok
4 ok
[seq 0: All nodes agree on 467994
[seq 1: All nodes agree on 85733
[seq 2: All nodes agree on 298006
[seq 3: All nodes agree on 235202
[seq 4: All nodes agree on 54942
[pass test before a few server die
Server shutdown...
Server shutdown...
server 3 and server 4 is down
seq 5: All nodes agree on 314324
seq 6: All nodes agree on 426333
alive servers function as expected
Server shutdown...
Server shutdown...
Server shutdown...
---------------Test_unreliable: Passed
```

- Test - Test minority

```
// make a majority of server die
std::set<int> deadServer = {2,3,4};
for (auto i : deadServer) {
  pxs.at(i)->TerminateService();
}
std::cout << "server 2 and server 3 and server 4 is down"

pxs.at(0)->Start(0,"0");
pxs.at(1)->Start(1,"1");
wait_fail(0, 1, pxs, peers_num);

std::cout << "minority servers cannot make any progress"
```

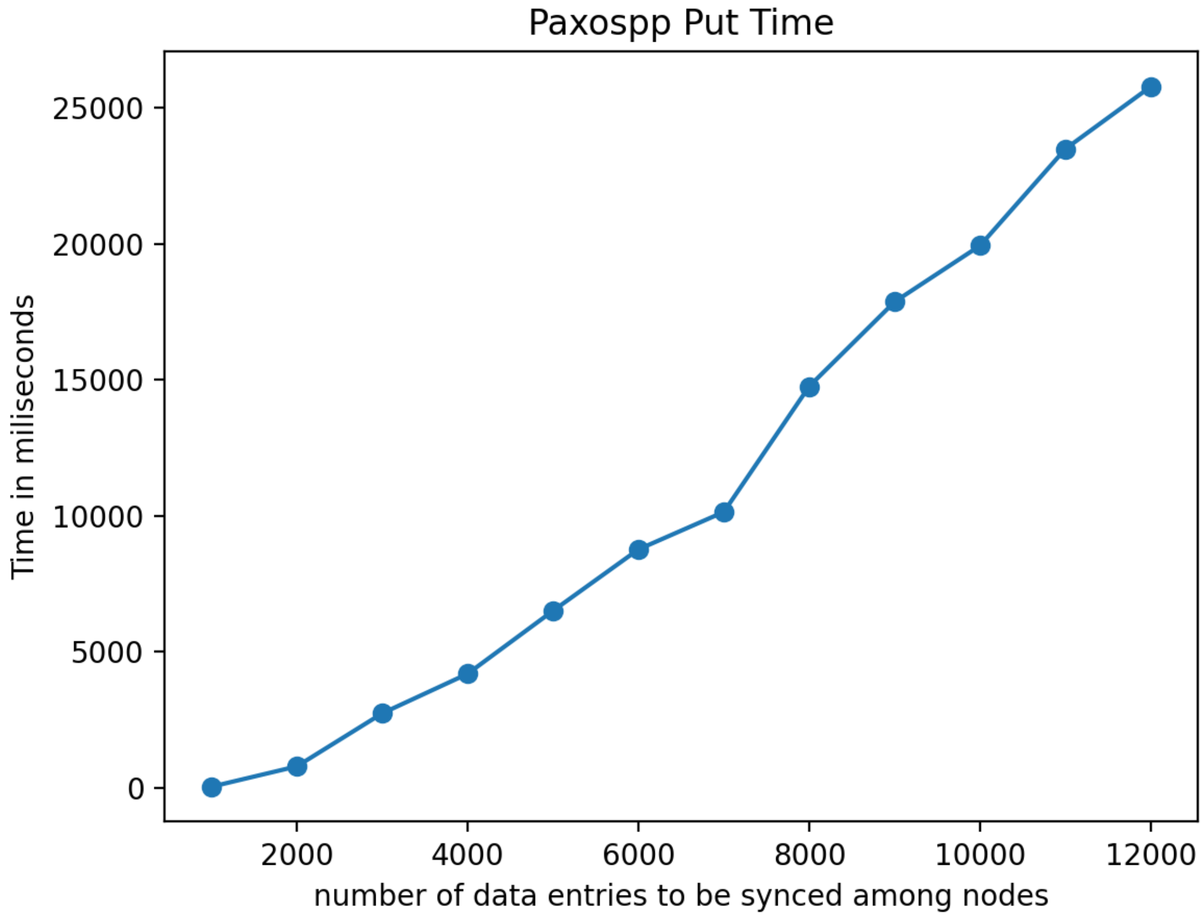
```
Luofeis-MBP:build zhangluofei$ ./test
0 ok
1 ok
2 ok
3 ok
4 ok
Server shutdown...
Server shutdown...
Server shutdown...
server 2 and server 3 and server 4 is down
seq 0: Too few servers alive, no agreement reached
seq 1: Too few servers alive, no agreement reached
minority servers cannot make any progress
Server shutdown...
Server shutdown...
----------------Test_minority: Passed
```

- Test - Concurrent

```
pxs.at(1)->Start(1,"hi");
pxs.at(2)->Start(1, "2");
pxs.at(0)->Start(1,"hello");

pxs.at(1)->Start(2,"hi2");
pxs.at(2)->Start(2, "22");
pxs.at(0)->Start(2,"hello2");

pxs.at(1)->Start(1,"1");

wait_check(1,2, pxs, peers_num);
```

```
Luofeis-MBP:build zhangluofei$ ./test
0 ok
1 ok
2 ok
3 ok
4 ok
seq 1: All nodes agree on 2
seq 2: All nodes agree on hello2
Server shutdown...
Server shutdown...
Server shutdown...
Server shutdown...
Server shutdown...
----------------Test_concurrent: Passed
```

Performance



- Time increases linearly along with the number of data entries
- Previous requests will not block later requests

Example Project (v1.2)

- **KVStore service**
  - A simple KVStore lib based on our implementation of Paxospp
  - A good illustration of why Paxos is useful
  - 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;
}
```

- KVStoreServer Class & API

```
class KVStoreServer final : public KVStore::Service {
  public:
    KVStoreServer(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);

    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);
    std::tuple<std::string, std::string> Put(const std::string& key, const std::string& value);
    std::tuple<std::string, std::string> Get(const std::string& key, const std::string& value);

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

- KVStoreServer Function Details

```
/* Put service for putting a value into db */
grpc::Status KVStoreServer::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;
}
```

- KVStoreServer Function Details

```
std::tuple<std::string, std::string> KVStoreServer::write_log(Op op)
{
    std::string op_str = encode(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", "");
      }
    }

    //...
}
```

- KVStoreServer Function Details

```
std::tuple<std::string, std::string> KVStoreServer::write_log(Op op)
{
    //...

    for (;;) {
      int seq = committed_seq + 1;
      std::cout << "KV server chose seq " << seq << "\n";
      px.Start(seq, op_str);
      Op returned_op = get_log(seq);
      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", "");
}
```

- KVStoreServer Function Details

```
std::tuple<std::string, std::string> KVStoreServer::write_log(Op op)
{
    //...

    for (;;) {
      int seq = committed_seq + 1;
      std::cout << "KV server chose seq " << seq << "\n";
      px.Start(seq, op_str);
      Op returned_op = get_log(seq);
      auto [ err, value ] = execute_log(returned_op);
      latest_requests[returned_op.client_id] = new Response
      committed_seq++;
      if (returned_op.client_id == op.client_id && returned
        return std::make_tuple(err, value);
      }
    }

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

```
Op KVStoreServer::get_log(int seq)
{
  bool end = false;
  std::string op_str;
  for (;!end;) {
    auto [decided, val] = px.Status(seq);
    end = decided;
    op_str = val;
    usleep(200);
  }
  Op op = decode(op_str);
  return op;
}
```

KVStore Demo


build — -bash — 138×41
...6/project/paxos/example/kvstore/cmake/build — -bash
...6/project/paxos/example/kvstore/cmake/build — -bash
...project/paxos/example/kvstore/cmake/build — -bash
...project/paxos/example/kvstore/cmake/build — -bash
pvs-MacBook-Pro:build pv$ ./kvstore-server --paxos 0.0.0.0:50051 0.0.0.0:50052 0.0.0.0:50053 -i 0 --address 0.0.0.0:50060

Limitation & Future Work

- Limitation on the laptop -> testing on other machines
- Flexibility of the data stored: std::string -> std::any
- Cleanup old enough local data when not needed
  - User would choose old time
- Develop services like distributed lock and register service based on our Paxos library

Reference

- https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
- https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf
- https://raft.github.io
- https://columbia.github.io/ds1-class/lectures/07-paxos.pdf
- https://columbia.github.io/ds1-class/lectures/07-paxos-functioning-slides.pdf