Branch: master ▾

Find file Copy path

paxosstore / paxoskv / core / **plog_wrapper.cc**

dengoswei - pass pins_wrapper_test;

1307944 on Aug 27, 2017

1 contributor

Raw Blame History

558 lines (476 sloc) 16 KB

```

/*
* Tencent is pleased to support the open source community by making PaxosStore available.
* Copyright (C) 2017 THL A29 Limited, a Tencent company. All rights reserved.
* Licensed under the BSD 3-Clause License (the "License"); you may not use this file except in compliance with the Licens
* https://opensource.org/licenses/BSD-3-Clause
* Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an
*/



#include <cassert>
#include "plog_wrapper.h"
#include "paxos.pb.h"
#include "pins_wrapper.h"
#include "plog_helper.h"
#include "cutils/mem_utils.h"
#include "cutils/log_utils.h"
#include "cutils/hassert.h"
#include "cutils/id_utils.h"

namespace {

std::unique_ptr<paxos::Message>
buildNoopRspMsg(const paxos::Message& req_msg, const uint64_t max_index)
{
    auto rsp_msg = cutils::make_unique<paxos::Message>();
    assert(nullptr != rsp_msg);

    rsp_msg->set_type(paxos::MessageType::NOOP);
    rsp_msg->set_from(req_msg.to());
    rsp_msg->set_to(req_msg.from());
    rsp_msg->set_key(req_msg.key());
    rsp_msg->set_index(max_index);
    rsp_msg->set_proposed_num(0);
    return rsp_msg;
}

bool belong_to(uint64_t reqid, uint16_t member_id)
{
    if (0 == reqid) {
        return false;
    }

    uint16_t req_member_id = 0;
    uint16_t req_cnt = 0;
    std::tie(req_member_id, req_cnt) =
        cutils::IDGenerator::decompose(reqid);
    return req_member_id == member_id;
}

} // namespace
```

```cpp

namespace paxos {


PLogWrapper::PLogWrapper(
        uint8_t selfid,
        uint16_t member_id,
        const std::string& key,
        PInsAliveState* pins_state,
        PaxosLog& plog_impl)
    : selfid_(selfid)
    , member_id_(member_id)
    , key_(key)
    , pins_state_(pins_state)
    , plog_impl_(plog_impl)
{
    assert(0 < selfid_);
    assert(is_slim(plog_impl_));

    if (0 < plog_impl_.entries_size()) {
        assert(2 >= plog_impl_.entries_size());
        auto min_index = get_min_index(plog_impl_);
        auto max_index = get_max_index(plog_impl_);
        assert(min_index == max_index || min_index + 1 == max_index);

        uint64_t index_sofar = 0;
        for (int idx = 0; idx < plog_impl_.entries_size(); ++idx) {
            const auto& ins = plog_impl_.entries(idx);
            if (ins.chosen()) {
                assert(ins.has_promised_num());
                assert(ins.has_accepted_num());
                assert(ins.has_accepted_value());
            }

            assert(0 == ins.index() || index_sofar < ins.index());
            index_sofar = ins.index();
        }
    }
}

PLogWrapper::~PLogWrapper() = default;

std::tuple<int, std::unique_ptr<PInsWrapper>>
PLogWrapper::getInstance(const uint64_t msg_index)
{
    assert(0 < msg_index);
    auto min_index = get_min_index(plog_impl_);
    auto max_index = get_max_index(plog_impl_);
    assert(min_index <= max_index);

    if (msg_index != min_index && msg_index != max_index) {
        if (msg_index < max_index) {
            return std::make_tuple(-1, nullptr);
        }

        assert(msg_index > max_index);
        // create a new pending ins
        // => drop prev ins
        if (nullptr != pins_state_) {
            pins_state_->SendNotify();
            pins_state_ = nullptr;
        }

        assert(nullptr == pins_state_);

        auto ins = plog_impl_.add_entries();
        assert(nullptr != ins);
        ins->set_index(msg_index);
    }

    paxos::PaxosInstance* ins = nullptr;
    for (int idx = 0; idx < plog_impl_.entries_size(); ++idx) {
        assert(nullptr != plog_impl_.mutable_entries(idx));
        if (msg_index != plog_impl_.entries(idx).index()) {
            continue;
        }

        ins = plog_impl_.mutable_entries(idx);
        assert(nullptr != ins);
        break;
    }

    assert(nullptr != ins);
    assert(msg_index == ins->index());
    return std::make_tuple(
            0, cutils::make_unique<PInsWrapper>(pins_state_, *ins));
}

std::tuple<int, std::unique_ptr<Message>>
        PLogWrapper::stepInvalidIndex(const Message& msg)
{
    assert(is_slim(plog_impl_));
    const int entries_size = plog_impl_.entries_size();
    if (0 == entries_size) {
        return std::make_tuple(0, nullptr);
    }

    auto max_index = get_max_index(plog_impl_);
    assert(0 < max_index && msg.index() < max_index);

        std::unique_ptr<Message> rsp_msg;
        int err = 0;
        std::unique_ptr<PInsWrapper> pins = nullptr;
        switch (msg.type())
        {
        case MessageType::GET_CHOSEN:
    case MessageType::PROP:
    case MessageType::ACCPT:
    case MessageType::FAST_ACCPT:
                {
            auto chosen_ins = get_chosen_ins(plog_impl_);
            if (nullptr == chosen_ins) {
                break; // do nothing
            }

            assert(nullptr != chosen_ins);
            assert(msg.index() < chosen_ins->index());

                        Message fake_msg = msg;
            fake_msg.set_type(MessageType::GET_CHOSEN);
                        fake_msg.set_index(chosen_ins->index());

                        std::tie(err, pins) = getInstance(fake_msg.index());
                        assert(0 == err);
                        assert(nullptr != pins);
                        bool write = false;
            // pins => chosen_ins !!!
                        std::tie(err, write, rsp_msg) = pins->Step(fake_msg);
                        assert(0 == err);
                        assert(false == write);
                        assert(nullptr != rsp_msg);
                        assert(MessageType::CHOSEN == rsp_msg->type());
```

```
                         assert(chosen_ins->index() == rsp_msg->index());
                         rsp_msg->set_to(msg.from());
                 }
                 break;
         case MessageType::CHOSEN:
                 {
             if (msg.index() + 1 != max_index ||
                     plog_impl_.entries(entries_size-1).chosen()) {
                                 break; // do nothing;
                         }

             assert(msg.index() + 1 == max_index);
             assert(false == plog_impl_.entries(entries_size-1).chosen());
             assert(1 == entries_size);

             PaxosInstance new_ins;
             new_ins.set_index(msg.index());
             pins = cutils::make_unique<PInsWrapper>(nullptr, new_ins);
             assert(nullptr != pins);

             PaxosLog plog_new;
             {
                 auto add_ins = plog_new.add_entries();
                 assert(nullptr != add_ins);
                 add_ins->Swap(&new_ins);

                 add_ins = plog_new.add_entries();
                 assert(nullptr != add_ins);
                 add_ins->Swap(plog_impl_.mutable_entries(entries_size-1));
             }

             plog_new.Swap(&plog_impl_);
             assert(2 == plog_impl_.entries_size());
             assert(is_slim(plog_impl_));
             setDiskWrite();
                         setUpdateChosen();
                 }
                 break;
         default:
                 break;
         }

         return std::make_tuple(0, std::move(rsp_msg));
}

std::tuple<int, std::unique_ptr<Message>>
    PLogWrapper::Step(const Message& msg)
{
    if ((0 == msg.index()) ||
            key_ != msg.key() ||
            static_cast<uint32_t>(selfid_) != msg.to()) {

                // GET_CHOSEN: fix case;
        if (0 == msg.index() && MessageType::GET_CHOSEN == msg.type()) {
            return stepInvalidIndex(msg);
        }

                logerr("msg.index %" PRIu64 " selfid %d msg.to %u",
                                msg.index(),
                                static_cast<int>(selfid_), msg.to());
        return std::make_tuple(-1, nullptr);
    }

    std::unique_ptr<Message> rsp_msg = nullptr;
    {
        bool write = false;
        int err = 0;
        std::unique_ptr<PInsWrapper> pins = nullptr;
        // may update commited_index;
        std::tie(err, pins) = getInstance(msg.index());
        if (0 != err) {
            assert(nullptr == pins);
                        assert(-1 == err);
            // msg_index < std::max(chosen_index, pending_index);
                        return stepInvalidIndex(msg);
        }

        assert(0 == err);
        assert(nullptr != pins);
        const bool already_chosen = pins->IsChosen();
        // - rsp msg;
        // - chosen ?
        //   => chosen_ins = pending_ins; pending_ins.clear();
                std::tie(err, write, rsp_msg) = pins->Step(msg);
                if (0 != err) {
                        assert(false == write);
                        assert(nullptr == rsp_msg);
                        return std::make_tuple(err, nullptr);
                }

        const bool now_chosen = pins->IsChosen();
        pins = nullptr;
        /*
                logdebug("key %" PRIu64 " %" PRIu64 " already_chosen %d now_chosen %d"
                                " reqmsgtype %d rsp_msg %p rsp_msg_type %d",
                                msg.logid(), msg.index(), already_chosen, now_chosen,
                                static_cast<int>(msg.type()),
                                rsp_msg.get(), nullptr == rsp_msg ? -1 : static_cast<int>(rsp_msg->type()));
        */
        if (false == already_chosen && now_chosen) {
            setDiskWrite();
        }

        assert(nullptr == pins);
        if (write) {
            setDiskWrite();
        }

        auto do_shrink = shrink_plog(plog_impl_);
        if (1 == do_shrink) {
            setDiskWrite();
        }
    }

    if (nullptr != rsp_msg) {
        assert(rsp_msg->index() == msg.index());
        assert(rsp_msg->key() == msg.key());
        assert(rsp_msg->from() == msg.to());
        assert(rsp_msg->from() == static_cast<uint32_t>(selfid_));
        assert(rsp_msg->key() == key_);
    }

    return std::make_tuple(0, std::move(rsp_msg));
}

std::tuple<
    int,
    std::shared_ptr<PInsAliveState>,
    std::unique_ptr<Message>>
PLogWrapper::Set(
        uint64_t reqid,
        const std::string& raw_value,
        bool do_fast_accpt)
{
    if (nullptr != pins_state_) {
        return std::make_tuple(-10, nullptr, nullptr);
    }

    assert(nullptr == pins_state_);
    auto max_ins = get_max_ins(plog_impl_);
    if (nullptr != max_ins && false == max_ins->chosen()) {
        return PreemptSet(reqid, raw_value);
    }

    assert(nullptr == max_ins || max_ins->chosen());
    return NormalSet(reqid, raw_value, do_fast_accpt);
}

std::tuple<
    int,
    std::shared_ptr<PInsAliveState>,
    std::unique_ptr<Message>>
PLogWrapper::NormalSet(
                uint64_t reqid,
        const std::string& data, const bool do_fast_accpt)
{
    assert(is_slim(plog_impl_));
    if (nullptr != pins_state_) {
        return std::make_tuple(-10, nullptr, nullptr);
    }

    assert(nullptr == pins_state_);
    auto max_ins = get_max_ins(plog_impl_);
    if (nullptr != max_ins && false == max_ins->chosen()) {
        return std::make_tuple(ErrorCode::BUSY, nullptr, nullptr);
    }

    assert(nullptr == max_ins || max_ins->chosen());
    uint64_t propose_index =
        nullptr == max_ins ? 1 : max_ins->index() + 1;

    Message msg;
    msg.set_type(MessageType::BEGIN_PROP);
    msg.set_from(selfid_);
    msg.set_to(selfid_);
    msg.set_key(key_);
    msg.set_index(propose_index);
    {
        auto entry = msg.mutable_accepted_value();
        assert(nullptr != entry);
        entry->set_reqid(reqid);
        entry->set_data(data);
    }

        // must be the case
        // assert(propose_index == pins_state_->GetIndex());
        // assert(PropState::NIL == pins_state_->GetPropState());
    bool can_do_fast = false;
    if (nullptr != max_ins) {
        assert(max_ins->chosen());
        if (belong_to(max_ins->accepted_value().reqid(), member_id_)) {
            can_do_fast = true;
        }
    }

        if (do_fast_accpt && can_do_fast) {
                msg.set_type(MessageType::BEGIN_FAST_PROP);
        }

    msg.set_proposed_num(
            cutils::prop_num_compose(selfid_, 0));

    auto shared_pins_state =
        std::make_shared<PInsAliveState>(
                key_, propose_index, msg.proposed_num());
    assert(nullptr != shared_pins_state);
    pins_state_ = shared_pins_state.get();
        // assert(0 == cutils::get_prop_cnt(pins_state_->GetProposedNum()));

    auto new_ins = plog_impl_.add_entries();
    assert(nullptr != new_ins);
    new_ins->set_index(propose_index);

    int ret = 0;
    std::unique_ptr<Message> rsp_msg;
    std::tie(ret, rsp_msg) = Step(msg);
    if (0 != ret) {
        pins_state_ = nullptr;
        return std::make_tuple(ret, nullptr, nullptr);
    }

    assert(0 == ret);
    assert(nullptr != rsp_msg);
    assert(shared_pins_state->GetIndex() == rsp_msg->index());
    return std::make_tuple(
            0, std::move(shared_pins_state), std::move(rsp_msg));
}

std::tuple<
    int,
    std::shared_ptr<PInsAliveState>,
    std::unique_ptr<Message>>
PLogWrapper::PreemptSet(uint64_t reqid, const std::string& data)
{
    assert(is_slim(plog_impl_));
        if (nullptr != pins_state_) {
                return std::make_tuple(-10, nullptr, nullptr);
        }

        // must be
        assert(nullptr == pins_state_);
    auto max_ins = get_max_ins(plog_impl_);
    auto chosen_ins = get_chosen_ins(plog_impl_);
    if (nullptr == max_ins ||
            nullptr == chosen_ins ||
            max_ins->index() != chosen_ins->index() + 1) {
        if (!(nullptr != max_ins && 1 == max_ins->index())) {
            return std::make_tuple(-11, nullptr, nullptr);
        }
    }

    assert(nullptr != max_ins);
    assert(false == max_ins->chosen());
    uint64_t propose_index = max_ins->index();
    assert(0 < propose_index);

        Message msg;
        msg.set_type(MessageType::TRY_PROP);
        msg.set_from(selfid_);
        msg.set_to(selfid_);
    msg.set_key(key_);
        msg.set_index(propose_index);
        msg.set_proposed_num(
                        cutils::PropNumGen(
                selfid_, 0).Next(max_ins->proposed_num()));
        hassert(msg.proposed_num() > max_ins->proposed_num(),
                        "msg.proposed_num %" PRIu64
            " max_ins.proposed_num %" PRIu64,
                        msg.proposed_num(), max_ins->proposed_num());
        {
                auto entry = msg.mutable_accepted_value();
                assert(nullptr != entry);
                entry->set_reqid(reqid);
                entry->set_data(data);
        }

    auto shared_pins_state =
        std::make_shared<PInsAliveState>(
                key_, propose_index, msg.proposed_num());
    assert(nullptr != shared_pins_state);
    pins_state_ = shared_pins_state.get();

    int ret = 0;
    std::unique_ptr<Message> rsp_msg;
    std::tie(ret, rsp_msg) = Step(msg);
    if (0 != ret) {
        pins_state_ = nullptr;
        return std::make_tuple(ret, nullptr, nullptr);
    }

    assert(0 == ret);
    assert(nullptr != rsp_msg);
    assert(shared_pins_state->GetIndex() == rsp_msg->index());
    return std::make_tuple(
            0, std::move(shared_pins_state), std::move(rsp_msg));
}

std::tuple<int, std::unique_ptr<Message>>
PLogWrapper::TryRedoProp()
{
    assert(is_slim(plog_impl_));
        if (nullptr == pins_state_) {
                return std::make_tuple(-20, nullptr);
        }

    assert(nullptr != pins_state_);
    auto max_ins = get_max_ins(plog_impl_);
    if (nullptr == max_ins || max_ins->chosen()) {
                return std::make_tuple(1, nullptr);
        }

    assert(nullptr != max_ins && false == max_ins->chosen());
    assert(pins_state_->GetIndex() == max_ins->index());

        Message msg;
        msg.set_type(MessageType::TRY_REDO_PROP);
        msg.set_from(selfid_);
        msg.set_to(selfid_);
    msg.set_key(key_);
        msg.set_index(max_ins->index());
        msg.set_proposed_num(
                        cutils::PropNumGen(selfid_, 0).Next(max_ins->proposed_num()));
        hassert(msg.proposed_num() > max_ins->proposed_num(),
                        "msg.proposed_num %" PRIu64
            " max_ins->proposed_num %" PRIu64,
                        msg.proposed_num(), max_ins->proposed_num());

        auto entry = msg.mutable_accepted_value();
        assert(nullptr != entry);
        // case 1:
```

```
        if (pins_state_->HasProposingValue()) {
                entry->set_reqid(pins_state_->GetProposingValue().reqid());
                entry->set_data(pins_state_->GetProposingValue().data());
                return Step(msg);
        }

        assert(false == pins_state_->HasProposingValue());
        // case 2:
        if (max_ins->has_accepted_value()) {
                entry->set_reqid(max_ins->accepted_value().reqid());
                entry->set_data(max_ins->accepted_value().data());
                return Step(msg);
        }

        assert(false == max_ins->has_accepted_value());
        // case 3:
    auto chosen_ins = get_chosen_ins(plog_impl_);
    if (nullptr == chosen_ins ||
            chosen_ins->index() + 1 != max_ins->index()) {
                logerr("FAILED LOCAL OUT: chosen_ins.index %" PRIu64
                                " pending_ins.index %" PRIu64,
                                chosen_ins->index(), max_ins->index());
                return std::make_tuple(-21, nullptr);
        }

    assert(nullptr != chosen_ins &&
            chosen_ins->index() + 1 == max_ins->index());
        entry->set_reqid(0);
        entry->set_data(chosen_ins->accepted_value().data());
        return Step(msg);
}

PInsAliveState*
PLogWrapper::SetPInsAliveState(PInsAliveState* new_pins_state)
{
    std::swap(pins_state_, new_pins_state);
    return new_pins_state;
}


} // namespace paxos


```