Branch: **master** | Find file | Copy path

paxosstore / paxoskv / core / **pins_wrapper.cc**

dengoswei - pass pins_wrapper_test;

1307944 on Aug 27, 2017

1 contributor

Raw | Blame | History

966 lines (838 sloc) 29.8 KB

```cpp

/*
* Tencent is pleased to support the open source community by making PaxosStore avai
* 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 fil
* https://opensource.org/licenses/BSD-3-Clause
* Unless required by applicable law or agreed to in writing, software distributed u
*/



#include <unistd.h>
#include <cassert>
#include "pins_wrapper.h"
#include "cutils/mem_utils.h"
#include "cutils/log_utils.h"
#include "cutils/hassert.h"


namespace {

inline void set_accepted_value(
        std::unique_ptr<paxos::Message>& rsp_msg,
        const paxos::Entry& accepted_value)
{
    assert(nullptr != rsp_msg);
    auto entry = rsp_msg->mutable_accepted_value();
    assert(nullptr != entry);
    *entry = accepted_value;
        assert(rsp_msg->accepted_value().reqid() == accepted_value.reqid());
        assert(rsp_msg->accepted_value().data() == accepted_value.data());
}

void updateRspVotes(
        uint8_t peer_id,
        bool vote,
        std::map<uint8_t, bool>& rsp_votes)
{
    assert(0 < peer_id);

    if (rsp_votes.end() != rsp_votes.find(peer_id)) {
        assert(rsp_votes[peer_id] == vote);
        return ;
    }

    // else
    rsp_votes[peer_id] = vote;
}

std::tuple<int, int> countVotes(const std::map<uint8_t, bool>& votes)
{
    int true_cnt = 0;
    int false_cnt = 0;
    for (const auto& v : votes) {
        if (v.second) {
            ++true_cnt;
        } else {
            ++false_cnt;
        }
    }

    return std::make_tuple(true_cnt, false_cnt);
}

inline bool
updatePromised(uint64_t prop_num, paxos::PaxosInstance& pins_impl)
{
    if (pins_impl.has_promised_num() &&
            pins_impl.promised_num() > prop_num) {
        return false; // reject
    }

    pins_impl.set_promised_num(prop_num);
    return true;
}

bool updateAccepted(
        uint64_t prop_num,
        const paxos::Entry& prop_value,
        bool is_fast_accept,
        paxos::PaxosInstance& pins_impl)
{
    if (pins_impl.has_promised_num() &&
            pins_impl.promised_num() > prop_num) {
        return false; // reject
    }

    assert(false == pins_impl.has_promised_num() ||
            pins_impl.promised_num() <= prop_num);
    if (true == is_fast_accept) {
        if (pins_impl.has_accepted_num()) {
            // do fast accepted only when accepted_num haven't be set
            return false; // reject
        }

        assert(false == pins_impl.has_accepted_num());
    }

    assert(false == pins_impl.has_accepted_num() ||
            pins_impl.accepted_num() <= prop_num);
    pins_impl.set_promised_num(prop_num);
    pins_impl.set_accepted_num(prop_num);
    {
        auto entry = pins_impl.mutable_accepted_value();
        assert(nullptr != entry);
        *entry = prop_value;
                assert(pins_impl.accepted_value().reqid() == prop_value.reqid());
                assert(pins_impl.accepted_value().data() == prop_value.data());
    }

    return true;
}


} // namespace

namespace paxos {

// function for test

std::unique_ptr<PInsAliveState> PInsAliveState::TestClone()
{
        auto clone_pins_state =
                cutils::make_unique<PInsAliveState>(key_, index_, prop_num_gen_.Get
        assert(nullptr != clone_pins_state);
        assert(clone_pins_state->prop_num_gen_.Get() == prop_num_gen_.Get());

        clone_pins_state->prop_state_ = prop_state_;
        clone_pins_state->max_accepted_hint_num_ = max_accepted_hint_num_;
        clone_pins_state->max_hint_num_ = max_hint_num_;
        clone_pins_state->rsp_votes_ = rsp_votes_;
        if (nullptr != proposing_value_)
        {
                clone_pins_state->proposing_value_ =
                        cutils::make_unique<Entry>(*proposing_value_);
                assert(nullptr != clone_pins_state->proposing_value_);
        }

        return clone_pins_state;
}

// end of function for test

PInsAliveState::PInsAliveState(
        const std::string& key,
                uint64_t index,
                uint64_t proposed_num)
    : key_(key)
        , index_(index)
    , prop_num_gen_(proposed_num)
{
        assert(0 < index);
    assert(0 == pipe(pipes_));
    assert(0 <= pipes_[0]);
    assert(0 <= pipes_[1]);
}

PInsAliveState::~PInsAliveState()
{
        assert(0 <= pipes_[0]);
        assert(0 <= pipes_[1]);
        close(pipes_[1]);
        close(pipes_[0]);
        pipes_[0] = -1;
        pipes_[1] = -1;
}


void PInsAliveState::MarkChosen()
{
    prop_state_ = PropState::CHOSEN;
    rsp_votes_.clear();
    proposing_value_ = nullptr;
    assert(IsChosen());
}

void PInsAliveState::SendNotify() const
{
        assert(0 <= pipes_[0]);
        assert(0 <= pipes_[1]);
        char ch = 'p';
        assert(1 == write(pipes_[1], &ch, 1));
}


PropState
PInsAliveState::stepPrepareRsp(
        uint8_t peer_id,
        uint64_t peer_promised_num,
        uint64_t peer_accepted_num,
        const paxos::Entry* peer_accepted_value)
{
    assert(0 < peer_id);
    assert(PropState::WAIT_PREPARE == prop_state_);

        assert(nullptr != proposing_value_);
    uint64_t proposed_num = prop_num_gen_.Get();
    // CHECK proposed_num == peer_promised_num
    assert(proposed_num <= peer_promised_num);
    updateRspVotes(
            peer_id, proposed_num == peer_promised_num, rsp_votes_);

        if (nullptr != peer_accepted_value) {
                assert(0 < peer_accepted_num);
                if (peer_accepted_num >= max_accepted_hint_num_) {
                        if (peer_accepted_num > max_accepted_hint_num_) {
                                max_accepted_hint_num_ = peer_accepted_num;
                                *proposing_value_ = *peer_accepted_value;
                        }
                        else {
                                assert(proposing_value_->reqid() == peer_accepted_v
                                assert(proposing_value_->data() == peer_accepted_va
                        }
                }
        }

    // else => reject
    int promised_cnt = 0;
    int reject_cnt = 0;
    std::tie(promised_cnt, reject_cnt) = countVotes(rsp_votes_);
    if (reject_cnt >= major_cnt_) {
        // reject by majority
        return PropState::PREPARE;
    }
    else if (promised_cnt + 1 >= major_cnt_) {
        // +1 => including self-vote
```

```
            return PropState::ACCEPT;
        }

        return PropState::WAIT_PREPARE;
}

PropState
PInsAliveState::stepAcceptRsp(
            uint8_t peer_id,
            uint64_t peer_accepted_num,
            bool is_fast_accept_rsp)
{
        assert(0 < peer_id);
        assert(PropState::WAIT_ACCEPT == prop_state_);

        uint64_t proposed_num = prop_num_gen_.Get();
        updateRspVotes(peer_id, proposed_num == peer_accepted_num, rsp_votes_);

        int accept_cnt = 0;
        int reject_cnt = 0;
        std::tie(accept_cnt, reject_cnt) = countVotes(rsp_votes_);
        if (reject_cnt >= major_cnt_) {
            return PropState::PREPARE;
        }
        else if (accept_cnt + 1 >= major_cnt_) {
            return PropState::CHOSEN;
        }

        return PropState::WAIT_ACCEPT;
}

PropState PInsAliveState::stepTryPropose(
            uint64_t hint_proposed_num,
            const paxos::Entry& try_proposing_value)
{
        // delete prop_state_ = PropState::PREPARE;
        prop_num_gen_.Update(hint_proposed_num);
            active_prop_cnt_ = 0;
            max_accepted_hint_num_ = 0;
        if (nullptr == proposing_value_) {
            proposing_value_ = cutils::make_unique<Entry>(try_proposing_value);
        }
            else {
                    *proposing_value_ = try_proposing_value;
            }

        // else => nothing change..
        assert(nullptr != proposing_value_);

        rsp_votes_.clear();
    return PropState::PREPARE;
}

PropState PInsAliveState::stepBeginPropose(
        uint64_t hint_proposed_num,
        const paxos::Entry& proposing_value)
{
    assert(PropState::NIL == prop_state_);
    prop_num_gen_.Update(hint_proposed_num);
    // delete prop_state_ = PropState::PREPARE;
    assert(nullptr == proposing_value_);
    proposing_value_ = cutils::make_unique<Entry>(proposing_value);
    return PropState::PREPARE;
}

PropState PInsAliveState::beginPreparePhase(PaxosInstance& pins_impl)
{
    assert(PropState::PREPARE == prop_state_);
        assert(nullptr != proposing_value_);

    pins_impl.set_proposed_num(prop_num_gen_.Get());
        if (pins_impl.has_accepted_num() &&
                        max_accepted_hint_num_ < pins_impl.accepted_num()) {
                assert(pins_impl.has_accepted_value());
                max_accepted_hint_num_ = pins_impl.accepted_num();
                *proposing_value_ = pins_impl.accepted_value();
        }

    if (false == updatePromised(prop_num_gen_.Get(), pins_impl)) {
        // reject
        return PropState::PREPARE;
    }

    rsp_votes_.clear();
    return PropState::WAIT_PREPARE;
}

PropState PInsAliveState::beginAcceptPhase(PaxosInstance& pins_impl)
{
    assert(PropState::ACCEPT == prop_state_);
    assert(nullptr != proposing_value_);
    if (false == updateAccepted(
                prop_num_gen_.Get(), *proposing_value_, false, pins_impl)) {
        // reject
        return PropState::PREPARE;
    }
```

```

        // reject promised may bring max_accepted_hint_num_ > pins_impl.accepted_nu
        // pins_impl.accepted_num may < max_accepted_hint_num_;
        // assert(pins_impl.accepted_num() >= max_accepted_hint_num_);

    rsp_votes_.clear();
    return PropState::WAIT_ACCEPT;
}



std::tuple<bool, MessageType>
PInsAliveState::updatePropState(
        PropState next_prop_state, PaxosInstance& pins_impl)
{
    bool write = false;
    auto rsp_msg_type = MessageType::NOOP;
    prop_state_ = next_prop_state;
    switch (prop_state_) {
        case PropState::PROP_FROZEN:
                logerr("REACHE MAX_PROP_CNT %d => PROP_FROZEN", MAX_PROP_CNT);
                assert(false == write);
                SendNotify(); // it's safe here!!
                break;

    case PropState::WAIT_PREPARE:
    case PropState::WAIT_ACCEPT:
        // nothing
        break;

    case PropState::CHOSEN:
        rsp_msg_type = MessageType::CHOSEN;
        rsp_votes_.clear();
        proposing_value_ = nullptr;
        break;

    case PropState::PREPARE:
        {
                        ++active_prop_cnt_;
                        if (active_prop_cnt_ > MAX_PROP_CNT)
                        {
                                // MAX_PROP_CNT reached !!
                                return updatePropState(PropState::PROP_FROZEN, pins
                        }

            if (pins_impl.has_promised_num()) {
                prop_num_gen_.Next(pins_impl.promised_num());
            }
```

```

                prop_num_gen_.Update(
                                            std::max(max_hint_num_, max_accepted_hint_n
                hassert(prop_num_gen_.Get() > pins_impl.proposed_num(),
                        "prop_num_gen_.Get %" PRIu64
                        " pins_impl.proposed_num %" PRIu64,
                        prop_num_gen_.Get(), pins_impl.proposed_num());
                assert(prop_num_gen_.Get() > pins_impl.promised_num());
                auto new_state = beginPreparePhase(pins_impl);
                assert(PropState::WAIT_PREPARE == new_state);

                bool new_write = false;
                MessageType tmp_rsp_msg_type = MessageType::NOOP;
                std::tie(new_write, tmp_rsp_msg_type)
                    = updatePropState(PropState::WAIT_PREPARE, pins_impl);
                assert(false == new_write);
                assert(MessageType::NOOP == tmp_rsp_msg_type);
                assert(PropState::WAIT_PREPARE == prop_state_);
                rsp_msg_type = MessageType::PROP;
                write = true;
            }
            break;

        case PropState::ACCEPT:
            {
                auto new_state = beginAcceptPhase(pins_impl);
                if (PropState::PREPARE == new_state) {
                    return updatePropState(PropState::PREPARE, pins_impl);
                }

                assert(PropState::WAIT_ACCEPT == new_state);

                bool new_write = false;
                MessageType tmp_rsp_msg_type = MessageType::NOOP;
                std::tie(new_write, tmp_rsp_msg_type)
                    = updatePropState(PropState::WAIT_ACCEPT, pins_impl);
                assert(false == new_write);
                assert(MessageType::NOOP == tmp_rsp_msg_type);
                assert(PropState::WAIT_ACCEPT == prop_state_);
                rsp_msg_type = MessageType::ACCPT;
                write = true;
            }
            break;

        default:
            assert(false);
        }

```

```
    return std::make_tuple(write, rsp_msg_type);
}


std::tuple<bool, MessageType>
PInsAliveState::Step(const Message& msg, PaxosInstance& pins_impl)
{
    assert(key_ == msg.key());
        assert(index_ == msg.index());
        assert(PropState::CHOSEN != prop_state_);

    bool write = false;
    MessageType rsp_msg_type = MessageType::NOOP;
    switch (msg.type()) {
    case MessageType::PROP_RSP:
        {
                        assert(nullptr != proposing_value_);
            if (PropState::WAIT_PREPARE != prop_state_ ||
                    pins_impl.proposed_num() != msg.proposed_num()) {
                logdebug("msgtype::PROP_RSP "
                        " index %" PRIu64
                        " but ins in state %d"
                        " pins_impl.proposed_num %" PRIu64
                        " msg.proposed_num %" PRIu64,
                        msg.index(),
                        static_cast<int>(prop_state_),
                        pins_impl.proposed_num(),
                        msg.proposed_num());
                break;
            }

            assert(PropState::WAIT_PREPARE == prop_state_);
                        PropState next_prop_state = PropState::NIL;
                        if (prop_num_gen_.Get() != msg.proposed_num())
                        {
                                // must be write failed
                                assert(prop_num_gen_.Get() > msg.proposed_num());
                                next_prop_state = PropState::PREPARE; // redo
                        }
                        else
                        {
                                assert(prop_num_gen_.Get() == msg.proposed_num());
                                assert(prop_num_gen_.Get() == pins_impl.proposed_nu
                                next_prop_state = stepPrepareRsp(
                                                msg.from(), msg.promised_num(), msg
                                                msg.has_accepted_value()
                                                        ? &msg.accepted_value() : n
                        }

            std::tie(write, rsp_msg_type)
                = updatePropState(next_prop_state, pins_impl);
            // valid check
            {
                if (MessageType::ACCPT == rsp_msg_type) {
                    assert(msg.proposed_num() == pins_impl.proposed_num());
                    assert(msg.proposed_num() == pins_impl.promised_num());
                    assert(msg.proposed_num() == pins_impl.accepted_num());
                }
            }
        }
        break;
    case MessageType::ACCPT_RSP:
    case MessageType::FAST_ACCPT_RSP:
        {
            assert(nullptr != proposing_value_);
            if (PropState::WAIT_ACCEPT != prop_state_
                    || pins_impl.proposed_num() != msg.proposed_num()) {
                logdebug("msg ACCPT_RSP index %" PRIu64
                        " but instance in state %d"
                        " pins_impl.proposed_num %" PRIu64
                        " msg.proposed_num %" PRIu64,
                        msg.index(),
                        static_cast<int>(prop_state_),
                        pins_impl.proposed_num(),
                        msg.proposed_num());
                break;
            }

            assert(PropState::WAIT_ACCEPT == prop_state_);
            assert(prop_num_gen_.Get() == msg.proposed_num());
            assert(prop_num_gen_.Get() == pins_impl.proposed_num());
            assert(msg.has_accepted_num());
            assert(false == msg.has_accepted_value());
            auto next_prop_state = stepAcceptRsp(
                    msg.from(), msg.accepted_num(),
                    MessageType::FAST_ACCPT_RSP == msg.type());

            // valid check
            {
                if (PropState::CHOSEN == next_prop_state) {
                    assert(msg.proposed_num() == pins_impl.proposed_num());
                    assert(msg.proposed_num() <= pins_impl.promised_num());
                    assert(msg.proposed_num() <= pins_impl.accepted_num());
                    // MUST BE:
                    // event if pins_impl.accepted_num > msg.proposed_num
                    hassert(proposing_value_->reqid() ==
                            pins_impl.accepted_value().reqid(),
                                                        "proposing_value_->reqid %"
                                                        " pins_impl.accepted_value(
                                                        proposing_value_->reqid(),
                                                        pins_impl.accepted_value().
                    assert(proposing_value_->data() ==
                            pins_impl.accepted_value().data());
                }
            }
            std::tie(write, rsp_msg_type)
                = updatePropState(next_prop_state, pins_impl);

            // update max_hint_num_
            if (msg.has_promised_num()) {
                max_hint_num_ = std::max(max_hint_num_, msg.promised_num());
            }

            if (msg.has_accepted_num()) {
                max_hint_num_ = std::max(max_hint_num_, msg.accepted_num());
            }
        }
        break;
        case MessageType::TRY_REDO_PROP:
    case MessageType::TRY_PROP:
        {
            PropState next_prop_state = PropState::NIL;

                        uint64_t hint_proposed_num = msg.proposed_num();
                        if (0 == cutils::get_prop_cnt(hint_proposed_num)) {
                                hint_proposed_num = cutils::prop_num_compose(0, 1);
                        }

                        assert(msg.has_accepted_value());
                        next_prop_state = stepTryPropose(
                                        hint_proposed_num, msg.accepted_value());
            assert(PropState::PREPARE == next_prop_state);

                        assert(0 == active_prop_cnt_);
                        assert(0 == max_accepted_hint_num_);

            std::tie(write, rsp_msg_type)
                = updatePropState(next_prop_state, pins_impl);
            assert(PropState::WAIT_PREPARE == prop_state_);
                        assert(rsp_votes_.empty());

                        active_begin_prop_num_ = pins_impl.proposed_num();
        }
        break;
    case MessageType::BEGIN_PROP:
    case MessageType::BEGIN_FAST_PROP:
        {
            // assert(0 == msg.proposed_num());
                        assert(nullptr == proposing_value_);
                        assert(0 == active_prop_cnt_);
                        assert(0 == max_accepted_hint_num_);
            // use msg.accepted_value as propose value
            assert(msg.has_accepted_value());
            if (pins_impl.has_promised_num()) {
                logerr("CONFLICT");
                break;
            }

            assert(PropState::NIL == prop_state_);
            assert(false == pins_impl.has_promised_num());
            assert(false == pins_impl.has_accepted_num());
            assert(false == pins_impl.has_accepted_value());

                        uint64_t hint_proposed_num = 0;
                        if (MessageType::BEGIN_PROP == msg.type()) {
                                hint_proposed_num = cutils::prop_num_compose(0, 1);
                        }

            auto next_prop_state = stepBeginPropose(
                    hint_proposed_num, msg.accepted_value());
            assert(PropState::PREPARE == next_prop_state);
            std::tie(write, rsp_msg_type)
                = updatePropState(next_prop_state, pins_impl);

            assert(true == write);
            assert(MessageType::PROP == rsp_msg_type);
            hassert(prop_num_gen_.Get() == pins_impl.proposed_num(),
                    "prop_num_gen_.Get %" PRIu64
                    " pins_impl.proposed_num %" PRIu64,
                    prop_num_gen_.Get(), pins_impl.proposed_num());
            assert(pins_impl.has_promised_num());
            assert(prop_num_gen_.Get() == pins_impl.promised_num());
            assert(false == pins_impl.has_accepted_num());
            assert(false == pins_impl.has_accepted_value());

            if (MessageType::BEGIN_FAST_PROP == msg.type()) {
                // fast prop
                // => skip prepare phase
                std::tie(write, rsp_msg_type)
                    = updatePropState(PropState::ACCEPT, pins_impl);
                assert(true == write);
                assert(MessageType::ACCPT == rsp_msg_type);
                assert(pins_impl.has_accepted_num());
                assert(pins_impl.has_accepted_value());
                rsp_msg_type = MessageType::FAST_ACCPT;
                                assert(0 == cutils::get_prop_cnt(pins_impl.proposed
            }

                        assert(0 == max_accepted_hint_num_);
                        assert(rsp_votes_.empty());
                        active_begin_prop_num_ = pins_impl.proposed_num();
        }
        break;
    default:
        assert(false);
    }

    return std::make_tuple(write, rsp_msg_type);
}

PInsWrapper::PInsWrapper(
        PInsAliveState* pins_state,
        PaxosInstance& pins_impl)
    : pins_state_(pins_state)
    , pins_impl_(pins_impl)
{
    if (pins_impl.chosen()) {
        assert(pins_impl.has_accepted_value());
    }
}

std::tuple<int, bool, std::unique_ptr<Message>>
    PInsWrapper::Step(const Message& msg)
{
    assert(msg.index() == pins_impl_.index());
        if (IsChosen()) {
                return stepChosen(msg);
        }

        assert(false == IsChosen());
        return stepNotChosen(msg);
}

std::tuple<int, bool, std::unique_ptr<Message>>
    PInsWrapper::stepChosen(const Message& msg)
{
    assert(true == IsChosen());
    bool write = false;
    std::unique_ptr<Message> rsp_msg = nullptr;

    assert(true == pins_impl_.has_promised_num());
    assert(true == pins_impl_.has_accepted_num());
    assert(true == pins_impl_.has_accepted_value());
    switch (msg.type()) {
    case MessageType::CHOSEN:
                // check
                if (msg.has_accepted_value())
                {
            // TODO:
                        if ((msg.accepted_value().data() !=
                                        pins_impl_.accepted_value().data()) ||
                                        (msg.accepted_value().reqid() !=
                                         pins_impl_.accepted_value().reqid()))
                        {
                                logerr("IMPORTANT INCONSISTENT index %" PRIu64
                                                " from %u to %u",
                                                msg.index(), msg.from(), msg.to());
                assert(false);
                        }
                }
    default:
        break;

        case MessageType::GET_CHOSEN:
    case MessageType::PROP:
        case MessageType::ACCPT:
    case MessageType::FAST_ACCPT:
        rsp_msg = cutils::make_unique<Message>();
        assert(nullptr != rsp_msg);

        rsp_msg->set_type(MessageType::CHOSEN);
        rsp_msg->set_index(msg.index());
        rsp_msg->set_key(msg.key());
        rsp_msg->set_from(msg.to());
        rsp_msg->set_to(msg.from());

        rsp_msg->set_proposed_num(pins_impl_.proposed_num());
        rsp_msg->set_promised_num(pins_impl_.promised_num());
        rsp_msg->set_accepted_num(pins_impl_.accepted_num());
                rsp_msg->set_timestamp(time(NULL));
        set_accepted_value(rsp_msg, pins_impl_.accepted_value());
        break;
    }

    assert(false == write);
    return std::make_tuple(0, write, move(rsp_msg));
}
```

```
void PInsWrapper::markChosen()
{
    pins_impl_.set_chosen(true);
    if (nullptr != pins_state_) {
        pins_state_->MarkChosen();
                assert(pins_state_->IsChosen());
    }
}

std::tuple<int, bool, std::unique_ptr<Message>>
    PInsWrapper::stepNotChosen(const Message& msg)
{
    assert(false == IsChosen());
    if (0 == access(
                "/home/qspace/data/kvsvr/plog_learner_only", F_OK)) {
                if (MessageType::CHOSEN != msg.type()) {
                        logerr("plog_learner_only msgtype %d",
                                        static_cast<int>(msg.type()));
                        return std::make_tuple(-50221, false, nullptr);
                }

                assert(MessageType::CHOSEN == msg.type());
        }

    bool write = false;
    MessageType rsp_msg_type = MessageType::NOOP;
    switch (msg.type()) {
    // for all
    case MessageType::NOOP:
        case MessageType::GET_CHOSEN:
        // do nothing
        break;

    case MessageType::CHOSEN:
        {
            // FOR NOW
            assert(true == msg.has_accepted_value());
            if (pins_impl_.has_accepted_num()
                    && msg.proposed_num() == pins_impl_.accepted_num()) {
                // mark already accepted entry as chosen
                assert(pins_impl_.has_accepted_value());
                // !! CHECK !!
                                if ((pins_impl_.accepted_value().reqid() !=
                                                        msg.accepted_value().reqid(
                                                (pins_impl_.accepted_value().data()
                                                 msg.accepted_value().data())) {
                                        logerr("IMPORTANT INCONSISTENT index %" PRI
                                                        " from %u to %u",
                                                    msg.index(), msg.from(), ms
                        assert(false);
                                    }
                    }
                    else {
                        // self roll promised, accepted, chosen
                        write = true;
                                    cutils::PropNumGen prop_num_gen(0, 100);
                                    uint64_t hint_num = std::max(
                                                msg.proposed_num(), pins_impl_.prom
                                    hint_num = std::max(hint_num, pins_impl_.proposed_n

                                    logimpt(" index %" PRIu64 " msg: proposed %" PRIu64
                                                " local: chosen_ %d promised %" PRI
                                                " hint_num %" PRIu64,
                                                msg.index(), msg.proposed_num(),
                              pins_impl_.chosen(),
                              pins_impl_.promised_num(),
                                                pins_impl_.accepted_num(),
                                                hint_num);

                                    auto chosen_prop_num = prop_num_gen.Next(hint_num);
                                    assert(chosen_prop_num > msg.proposed_num());
                                    assert(chosen_prop_num > pins_impl_.promised_num())
                                    assert(chosen_prop_num > pins_impl_.proposed_num())
                        pins_impl_.set_proposed_num(chosen_prop_num);
                        assert(updatePromised(chosen_prop_num, pins_impl_));
                        assert(updateAccepted(
                                    chosen_prop_num,
                                    msg.accepted_value(), false, pins_impl_));
                    }

                    markChosen();
                    // not rsp_msg;
                }
                break;

        // accepter
        case MessageType::PROP:
                {
                    if (updatePromised(msg.proposed_num(), pins_impl_)) {
                        // promised =>
                        write = true;
                    }
                    rsp_msg_type = MessageType::PROP_RSP;
                }
                break;

```

```
    case MessageType::ACCPT:
    case MessageType::FAST_ACCPT:
        {
            assert(msg.has_accepted_value());
            bool fast_accept = MessageType::FAST_ACCPT == msg.type();
            if (updateAccepted(
                        msg.proposed_num(),
                        msg.accepted_value(), fast_accept, pins_impl_)) {
                // accepted other
                write = true;
            }

            rsp_msg_type = fast_accept
                ? MessageType::FAST_ACCPT_RSP : MessageType::ACCPT_RSP;
        }
        break;

    // proposer
    case MessageType::PROP_RSP:
    case MessageType::ACCPT_RSP:
    case MessageType::FAST_ACCPT_RSP:
        // start a propose
    case MessageType::BEGIN_PROP:
    case MessageType::TRY_PROP:
    case MessageType::BEGIN_FAST_PROP:
        case MessageType::TRY_REDO_PROP: // new add:
        if (nullptr == pins_state_) {
            logdebug("pins_state nullptr but recv msgtype %d",
                    static_cast<int>(msg.type()));
            break; // just ignore all proposer releated msg
        }

        assert(nullptr != pins_state_);
        assert(msg.key() == pins_state_->GetKey());
                assert(msg.index() == pins_state_->GetIndex());
        std::tie(write, rsp_msg_type) = pins_state_->Step(msg, pins_impl_);
        break;

    default:
        assert(false);
        break;
    }

    auto rsp_msg = produceRsp(msg, rsp_msg_type);
    if (MessageType::CHOSEN == rsp_msg_type) {
        assert(nullptr != rsp_msg);
        assert(false == IsChosen());
        markChosen();
    }
    return std::make_tuple(0, write, std::move(rsp_msg));
}

std::unique_ptr<Message>
PInsWrapper::produceRsp(const Message& msg, MessageType rsp_msg_type)
{
    if (MessageType::NOOP == rsp_msg_type) {
        return nullptr;
    }

    std::unique_ptr<Message> rsp_msg = cutils::make_unique<Message>();
    assert(nullptr != rsp_msg);
    rsp_msg->set_key(msg.key());
    rsp_msg->set_index(msg.index());
    rsp_msg->set_from(msg.to());
    rsp_msg->set_type(rsp_msg_type);
    rsp_msg->set_proposed_num(msg.proposed_num());
        rsp_msg->set_timestamp(time(NULL));
    switch (rsp_msg_type) {

    // accepter
    case MessageType::PROP_RSP:
        rsp_msg->set_promised_num(pins_impl_.promised_num());
        assert(rsp_msg->promised_num() >= rsp_msg->proposed_num());

                // TODO: add test
                // promised or reject => both need send back accepted_num
                if (pins_impl_.has_accepted_num()) {
                        assert(pins_impl_.has_accepted_value());
                        rsp_msg->set_accepted_num(pins_impl_.accepted_num());
                        set_accepted_value(rsp_msg, pins_impl_.accepted_value());
                }

        rsp_msg->set_to(msg.from());
        break;

    case MessageType::ACCPT_RSP:
    case MessageType::FAST_ACCPT_RSP:
        {
            assert(pins_impl_.has_promised_num());
            // => reject ?
            // assert(pins_impl_.has_accepted_num());
            // assert(pins_impl_.has_accepted_value());

            // TODO: ? send back pins_impl_.promised_num as a hint
            auto accepted_num =
                pins_impl_.has_accepted_num() ? pins_impl_.accepted_num() : 0;
            rsp_msg->set_accepted_num(accepted_num);
            if (accepted_num != msg.proposed_num()) {
                // reject => return promised_num as a hint
                if (0 == accepted_num) {
                    rsp_msg->set_promised_num(pins_impl_.promised_num());
                }
            }
            rsp_msg->set_to(msg.from());
        }
        break;

    // proposer
    case MessageType::PROP:
        assert(nullptr != pins_state_);
        rsp_msg->set_proposed_num(pins_impl_.proposed_num());
        // set_to 0 => broad-cast
        rsp_msg->set_to(0);
                assert(0 < cutils::get_prop_cnt(rsp_msg->proposed_num()));
        break;

    case MessageType::ACCPT:
    case MessageType::FAST_ACCPT:
        assert(nullptr != pins_state_);
        assert(pins_impl_.has_promised_num());
        assert(pins_impl_.has_accepted_num());
        assert(pins_impl_.has_accepted_value());

                assert(pins_impl_.proposed_num() == pins_state_->GetProposedNum());
                assert(pins_state_->HasProposingValue());
                assert(pins_impl_.accepted_value().data() ==
                                pins_state_->GetProposingValue().data());
                assert(pins_impl_.accepted_value().reqid() ==
                                pins_state_->GetProposingValue().reqid());
                rsp_msg->set_proposed_num(pins_state_->GetProposedNum());
                set_accepted_value(rsp_msg, pins_state_->GetProposingValue());
        rsp_msg->set_to(0);

                // check
                if (MessageType::ACCPT == rsp_msg_type) {
                        assert(0 < cutils::get_prop_cnt(rsp_msg->proposed_num()));
                }
                else {
                        assert(MessageType::FAST_ACCPT == rsp_msg_type);
                        assert(0 == cutils::get_prop_cnt(rsp_msg->proposed_num()));
                }
        break;

    case MessageType::CHOSEN:
        assert(MessageType::CHOSEN != msg.type());
        assert(pins_impl_.has_promised_num());
        assert(pins_impl_.has_accepted_num());
        assert(pins_impl_.has_accepted_value());
        rsp_msg->set_proposed_num(pins_impl_.proposed_num());
        set_accepted_value(rsp_msg, pins_impl_.accepted_value());
        rsp_msg->set_to(0);
        break;
    default:
        assert(false);
        break;
    }

    assert(rsp_msg->from() == msg.to());
    return rsp_msg;
}

} // namespace paxos

```