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

```
1
2  /*
3  * Tencent is pleased to support the open source community by making PaxosStore avai
4  * Copyright (C) 2017 THL A29 Limited, a Tencent company. All rights reserved.
5  * Licensed under the BSD 3-Clause License (the "License"); you may not use this fil
6  * https://opensource.org/licenses/BSD-3-Clause
7  * Unless required by applicable law or agreed to in writing, software distributed u
8  */
9
10
11
12 #include <unistd.h>
13 #include <cassert>
14 #include "pins_wrapper.h"
15 #include "cutils/mem_utils.h"
16 #include "cutils/log_utils.h"
17 #include "cutils/hassert.h"
18
19
20 namespace {
21
22 inline void set_accepted_value(
23         std::unique_ptr<paxos::Message>& rsp_msg,
24         const paxos::Entry& accepted_value)
25 {
26     assert(nullptr != rsp_msg);
27     auto entry = rsp_msg->mutable_accepted_value();
28     assert(nullptr != entry);
29     *entry = accepted_value;
30         assert(rsp_msg->accepted_value().reqid() == accepted_value.reqid());
31         assert(rsp_msg->accepted_value().data() == accepted_value.data());
32 }
33
34 void updateRspVotes(
```

```
35          uint8_t peer_id,
36          bool vote,
37          std::map<uint8_t, bool>& rsp_votes)
38  {
39      assert(0 < peer_id);
40
41      if (rsp_votes.end() != rsp_votes.find(peer_id)) {
42          assert(rsp_votes[peer_id] == vote);
43          return ;
44      }
45
46      // else
47      rsp_votes[peer_id] = vote;
48  }
49
50  std::tuple<int, int> countVotes(const std::map<uint8_t, bool>& votes)
51  {
52      int true_cnt = 0;
53      int false_cnt = 0;
54      for (const auto& v : votes) {
55          if (v.second) {
56              ++true_cnt;
57          } else {
58              ++false_cnt;
59          }
60      }
61
62      return std::make_tuple(true_cnt, false_cnt);
63  }
64
65  inline bool
66  updatePromised(uint64_t prop_num, paxos::PaxosInstance& pins_impl)
67  {
68      if (pins_impl.has_promised_num() &&
69              pins_impl.promised_num() > prop_num) {
70          return false; // reject
71      }
72
73      pins_impl.set_promised_num(prop_num);
74      return true;
75  }
76
77  bool updateAccepted(
78          uint64_t prop_num,
79          const paxos::Entry& prop_value,
80          bool is_fast_accept,
81          paxos::PaxosInstance& pins_impl)
82  {
```

```
    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_;
```

```
131         clone_pins_state->rsp_votes_ = rsp_votes_;
132         if (nullptr != proposing_value_)
133         {
134                 clone_pins_state->proposing_value_ =
135                         cutils::make_unique<Entry>(*proposing_value_);
136                 assert(nullptr != clone_pins_state->proposing_value_);
137         }
138
139         return clone_pins_state;
140 }
141
142 // end of function for test
143
144 PInsAliveState::PInsAliveState(
145         const std::string& key,
146                 uint64_t index,
147                 uint64_t proposed_num)
148     : key_(key)
149         , index_(index)
150     , prop_num_gen_(proposed_num)
151 {
152         assert(0 < index);
153     assert(0 == pipe(pipes_));
154     assert(0 <= pipes_[0]);
155     assert(0 <= pipes_[1]);
156 }
157
158 PInsAliveState::~PInsAliveState()
159 {
160         assert(0 <= pipes_[0]);
161         assert(0 <= pipes_[1]);
162         close(pipes_[1]);
163         close(pipes_[0]);
164         pipes_[0] = -1;
165         pipes_[1] = -1;
166 }
167
168
169 void PInsAliveState::MarkChosen()
170 {
171     prop_state_ = PropState::CHOSEN;
172     rsp_votes_.clear();
173     proposing_value_ = nullptr;
174     assert(IsChosen());
175 }
176
177 void PInsAliveState::SendNotify() const
178 {
```

```
179         assert(0 <= pipes_[0]);
180         assert(0 <= pipes_[1]);
181         char ch = 'p';
182         assert(1 == write(pipes_[1], &ch, 1));
183 }
184
185
186 PropState
187 PInsAliveState::stepPrepareRsp(
188         uint8_t peer_id,
189         uint64_t peer_promised_num,
190         uint64_t peer_accepted_num,
191         const paxos::Entry* peer_accepted_value)
192 {
193     assert(0 < peer_id);
194     assert(PropState::WAIT_PREPARE == prop_state_);
195
196         assert(nullptr != proposing_value_);
197     uint64_t proposed_num = prop_num_gen_.Get();
198     // CHECK proposed_num == peer_promised_num
199     assert(proposed_num <= peer_promised_num);
200     updateRspVotes(
201             peer_id, proposed_num == peer_promised_num, rsp_votes_);
202
203         if (nullptr != peer_accepted_value) {
204                 assert(0 < peer_accepted_num);
205                 if (peer_accepted_num >= max_accepted_hint_num_) {
206                         if (peer_accepted_num > max_accepted_hint_num_) {
207                                 max_accepted_hint_num_ = peer_accepted_num;
208                                 *proposing_value_ = *peer_accepted_value;
209                         }
210                         else {
211                                 assert(proposing_value_->reqid() == peer_accepted_v
212                                 assert(proposing_value_->data() == peer_accepted_va
213                         }
214                 }
215         }
216
217     // else => reject
218     int promised_cnt = 0;
219     int reject_cnt = 0;
220     std::tie(promised_cnt, reject_cnt) = countVotes(rsp_votes_);
221     if (reject_cnt >= major_cnt_) {
222         // reject by majority
223         return PropState::PREPARE;
224     }
225     else if (promised_cnt + 1 >= major_cnt_) {
226         // +1 => including self-vote
```

```
227         return PropState::ACCEPT;
228     }
229
230     return PropState::WAIT_PREPARE;
231 }
232
233 PropState
234 PInsAliveState::stepAcceptRsp(
235         uint8_t peer_id,
236         uint64_t peer_accepted_num,
237         bool is_fast_accept_rsp)
238 {
239     assert(0 < peer_id);
240     assert(PropState::WAIT_ACCEPT == prop_state_);
241
242     uint64_t proposed_num = prop_num_gen_.Get();
243     updateRspVotes(peer_id, proposed_num == peer_accepted_num, rsp_votes_);
244
245     int accept_cnt = 0;
246     int reject_cnt = 0;
247     std::tie(accept_cnt, reject_cnt) = countVotes(rsp_votes_);
248     if (reject_cnt >= major_cnt_) {
249         return PropState::PREPARE;
250     }
251     else if (accept_cnt + 1 >= major_cnt_) {
252         return PropState::CHOSEN;
253     }
254
255     return PropState::WAIT_ACCEPT;
256 }
257
258 PropState PInsAliveState::stepTryPropose(
259         uint64_t hint_proposed_num,
260         const paxos::Entry& try_proposing_value)
261 {
262     // delete prop_state_ = PropState::PREPARE;
263     prop_num_gen_.Update(hint_proposed_num);
264         active_prop_cnt_ = 0;
265         max_accepted_hint_num_ = 0;
266     if (nullptr == proposing_value_) {
267         proposing_value_ = cutils::make_unique<Entry>(try_proposing_value);
268     }
269         else {
270                 *proposing_value_ = try_proposing_value;
271         }
272
273     // else => nothing change..
274     assert(nullptr != proposing_value_);
```

```
275
276         rsp_votes_.clear();
277     return PropState::PREPARE;
278 }
279
280 PropState PInsAliveState::stepBeginPropose(
281         uint64_t hint_proposed_num,
282         const paxos::Entry& proposing_value)
283 {
284     assert(PropState::NIL == prop_state_);
285     prop_num_gen_.Update(hint_proposed_num);
286     // delete prop_state_ = PropState::PREPARE;
287     assert(nullptr == proposing_value_);
288     proposing_value_ = cutils::make_unique<Entry>(proposing_value);
289     return PropState::PREPARE;
290 }
291
292 PropState PInsAliveState::beginPreparePhase(PaxosInstance& pins_impl)
293 {
294     assert(PropState::PREPARE == prop_state_);
295         assert(nullptr != proposing_value_);
296
297     pins_impl.set_proposed_num(prop_num_gen_.Get());
298         if (pins_impl.has_accepted_num() &&
299                         max_accepted_hint_num_ < pins_impl.accepted_num()) {
300                 assert(pins_impl.has_accepted_value());
301                 max_accepted_hint_num_ = pins_impl.accepted_num();
302                 *proposing_value_ = pins_impl.accepted_value();
303         }
304
305     if (false == updatePromised(prop_num_gen_.Get(), pins_impl)) {
306         // reject
307         return PropState::PREPARE;
308     }
309
310     rsp_votes_.clear();
311     return PropState::WAIT_PREPARE;
312 }
313
314 PropState PInsAliveState::beginAcceptPhase(PaxosInstance& pins_impl)
315 {
316     assert(PropState::ACCEPT == prop_state_);
317     assert(nullptr != proposing_value_);
318     if (false == updateAccepted(
319                 prop_num_gen_.Get(), *proposing_value_, false, pins_impl)) {
320         // reject
321         return PropState::PREPARE;
322     }
```

```
323
324         // reject promised may bring max_accepted_hint_num_ > pins_impl.accepted_nu
325         // pins_impl.accepted_num may < max_accepted_hint_num_;
326         // assert(pins_impl.accepted_num() >= max_accepted_hint_num_);
327
328     rsp_votes_.clear();
329     return PropState::WAIT_ACCEPT;
330 }
331
332
333
334 std::tuple<bool, MessageType>
335 PInsAliveState::updatePropState(
336         PropState next_prop_state, PaxosInstance& pins_impl)
337 {
338     bool write = false;
339     auto rsp_msg_type = MessageType::NOOP;
340     prop_state_ = next_prop_state;
341     switch (prop_state_) {
342         case PropState::PROP_FROZEN:
343                 logerr("REACHE MAX_PROP_CNT %d => PROP_FROZEN", MAX_PROP_CNT);
344                 assert(false == write);
345                 SendNotify(); // it's safe here!!
346                 break;
347
348     case PropState::WAIT_PREPARE:
349     case PropState::WAIT_ACCEPT:
350         // nothing
351         break;
352
353     case PropState::CHOSEN:
354         rsp_msg_type = MessageType::CHOSEN;
355         rsp_votes_.clear();
356         proposing_value_ = nullptr;
357         break;
358
359     case PropState::PREPARE:
360         {
361                         ++active_prop_cnt_;
362                         if (active_prop_cnt_ > MAX_PROP_CNT)
363                         {
364                                 // MAX_PROP_CNT reached !!
365                                 return updatePropState(PropState::PROP_FROZEN, pins
366                         }
367
368             if (pins_impl.has_promised_num()) {
369                 prop_num_gen_.Next(pins_impl.promised_num());
370             }
```

```
371
372             prop_num_gen_.Update(
373                                     std::max(max_hint_num_, max_accepted_hint_n
374             hassert(prop_num_gen_.Get() > pins_impl.proposed_num(),
375                     "prop_num_gen_.Get %" PRIu64
376                     " pins_impl.proposed_num %" PRIu64,
377                     prop_num_gen_.Get(), pins_impl.proposed_num());
378             assert(prop_num_gen_.Get() > pins_impl.promised_num());
379             auto new_state = beginPreparePhase(pins_impl);
380             assert(PropState::WAIT_PREPARE == new_state);
381
382             bool new_write = false;
383             MessageType tmp_rsp_msg_type = MessageType::NOOP;
384             std::tie(new_write, tmp_rsp_msg_type)
385                 = updatePropState(PropState::WAIT_PREPARE, pins_impl);
386             assert(false == new_write);
387             assert(MessageType::NOOP == tmp_rsp_msg_type);
388             assert(PropState::WAIT_PREPARE == prop_state_);
389             rsp_msg_type = MessageType::PROP;
390             write = true;
391         }
392         break;
393
394     case PropState::ACCEPT:
395         {
396             auto new_state = beginAcceptPhase(pins_impl);
397             if (PropState::PREPARE == new_state) {
398                 return updatePropState(PropState::PREPARE, pins_impl);
399             }
400
401             assert(PropState::WAIT_ACCEPT == new_state);
402
403             bool new_write = false;
404             MessageType tmp_rsp_msg_type = MessageType::NOOP;
405             std::tie(new_write, tmp_rsp_msg_type)
406                 = updatePropState(PropState::WAIT_ACCEPT, pins_impl);
407             assert(false == new_write);
408             assert(MessageType::NOOP == tmp_rsp_msg_type);
409             assert(PropState::WAIT_ACCEPT == prop_state_);
410             rsp_msg_type = MessageType::ACCPT;
411             write = true;
412         }
413         break;
414
415     default:
416         assert(false);
417     }
418
```

```
419     return std::make_tuple(write, rsp_msg_type);
420 }
421
422
423 std::tuple<bool, MessageType>
424 PInsAliveState::Step(const Message& msg, PaxosInstance& pins_impl)
425 {
426     assert(key_ == msg.key());
427         assert(index_ == msg.index());
428         assert(PropState::CHOSEN != prop_state_);
429
430     bool write = false;
431     MessageType rsp_msg_type = MessageType::NOOP;
432     switch (msg.type()) {
433     case MessageType::PROP_RSP:
434         {
435                         assert(nullptr != proposing_value_);
436             if (PropState::WAIT_PREPARE != prop_state_ ||
437                     pins_impl.proposed_num() != msg.proposed_num()) {
438                 logdebug("msgtype::PROP_RSP "
439                         " index %" PRIu64
440                         " but ins in state %d"
441                         " pins_impl.proposed_num %" PRIu64
442                         " msg.proposed_num %" PRIu64,
443                         msg.index(),
444                         static_cast<int>(prop_state_),
445                         pins_impl.proposed_num(),
446                         msg.proposed_num());
447                 break;
448             }
449
450             assert(PropState::WAIT_PREPARE == prop_state_);
451                         PropState next_prop_state = PropState::NIL;
452                         if (prop_num_gen_.Get() != msg.proposed_num())
453                         {
454                                 // must be write failed
455                                 assert(prop_num_gen_.Get() > msg.proposed_num());
456                                 next_prop_state = PropState::PREPARE; // redo
457                         }
458                         else
459                         {
460                                 assert(prop_num_gen_.Get() == msg.proposed_num());
461                                 assert(prop_num_gen_.Get() == pins_impl.proposed_nu
462                                 next_prop_state = stepPrepareRsp(
463                                                 msg.from(), msg.promised_num(), msg
464                                                 msg.has_accepted_value()
465                                                         ? &msg.accepted_value() : n
466                         }
```

```
467
468             std::tie(write, rsp_msg_type)
469                 = updatePropState(next_prop_state, pins_impl);
470             // valid check
471             {
472                 if (MessageType::ACCPT == rsp_msg_type) {
473                     assert(msg.proposed_num() == pins_impl.proposed_num());
474                     assert(msg.proposed_num() == pins_impl.promised_num());
475                     assert(msg.proposed_num() == pins_impl.accepted_num());
476                 }
477             }
478         }
479         break;
480     case MessageType::ACCPT_RSP:
481     case MessageType::FAST_ACCPT_RSP:
482         {
483             assert(nullptr != proposing_value_);
484             if (PropState::WAIT_ACCEPT != prop_state_
485                     || pins_impl.proposed_num() != msg.proposed_num()) {
486                 logdebug("msg ACCPT_RSP index %" PRIu64
487                         " but instance in state %d"
488                         " pins_impl.proposed_num %" PRIu64
489                         " msg.proposed_num %" PRIu64,
490                         msg.index(),
491                         static_cast<int>(prop_state_),
492                         pins_impl.proposed_num(),
493                         msg.proposed_num());
494                 break;
495             }
496
497             assert(PropState::WAIT_ACCEPT == prop_state_);
498             assert(prop_num_gen_.Get() == msg.proposed_num());
499             assert(prop_num_gen_.Get() == pins_impl.proposed_num());
500             assert(msg.has_accepted_num());
501             assert(false == msg.has_accepted_value());
502             auto next_prop_state = stepAcceptRsp(
503                     msg.from(), msg.accepted_num(),
504                     MessageType::FAST_ACCPT_RSP == msg.type());
505
506             // valid check
507             {
508                 if (PropState::CHOSEN == next_prop_state) {
509                     assert(msg.proposed_num() == pins_impl.proposed_num());
510                     assert(msg.proposed_num() <= pins_impl.promised_num());
511                     assert(msg.proposed_num() <= pins_impl.accepted_num());
512                     // MUST BE:
513                     // event if pins_impl.accepted_num > msg.proposed_num
514                     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;
```

```
563     case MessageType::BEGIN_PROP:
564     case MessageType::BEGIN_FAST_PROP:
565         {
566             // assert(0 == msg.proposed_num());
567                         assert(nullptr == proposing_value_);
568                         assert(0 == active_prop_cnt_);
569                         assert(0 == max_accepted_hint_num_);
570             // use msg.accepted_value as propose value
571             assert(msg.has_accepted_value());
572             if (pins_impl.has_promised_num()) {
573                 logerr("CONFLICT");
574                 break;
575             }
576 
577             assert(PropState::NIL == prop_state_);
578             assert(false == pins_impl.has_promised_num());
579             assert(false == pins_impl.has_accepted_num());
580             assert(false == pins_impl.has_accepted_value());
581 
582                         uint64_t hint_proposed_num = 0;
583                         if (MessageType::BEGIN_PROP == msg.type()) {
584                                 hint_proposed_num = cutils::prop_num_compose(0, 1);
585                         }
586 
587             auto next_prop_state = stepBeginPropose(
588                     hint_proposed_num, msg.accepted_value());
589             assert(PropState::PREPARE == next_prop_state);
590             std::tie(write, rsp_msg_type)
591                 = updatePropState(next_prop_state, pins_impl);
592 
593             assert(true == write);
594             assert(MessageType::PROP == rsp_msg_type);
595             hassert(prop_num_gen_.Get() == pins_impl.proposed_num(),
596                     "prop_num_gen_.Get %" PRIu64
597                     " pins_impl.proposed_num %" PRIu64,
598                     prop_num_gen_.Get(), pins_impl.proposed_num());
599             assert(pins_impl.has_promised_num());
600             assert(prop_num_gen_.Get() == pins_impl.promised_num());
601             assert(false == pins_impl.has_accepted_num());
602             assert(false == pins_impl.has_accepted_value());
603 
604             if (MessageType::BEGIN_FAST_PROP == msg.type()) {
605                 // fast prop
606                 // => skip prepare phase
607                 std::tie(write, rsp_msg_type)
608                     = updatePropState(PropState::ACCEPT, pins_impl);
609                 assert(true == write);
610                 assert(MessageType::ACCPT == rsp_msg_type);
```

```
611                 assert(pins_impl.has_accepted_num());
612                 assert(pins_impl.has_accepted_value());
613                 rsp_msg_type = MessageType::FAST_ACCPT;
614                                 assert(0 == cutils::get_prop_cnt(pins_impl.proposed
615             }
616
617                         assert(0 == max_accepted_hint_num_);
618                         assert(rsp_votes_.empty());
619                         active_begin_prop_num_ = pins_impl.proposed_num();
620         }
621         break;
622     default:
623         assert(false);
624     }
625
626     return std::make_tuple(write, rsp_msg_type);
627 }
628
629 PInsWrapper::PInsWrapper(
630         PInsAliveState* pins_state,
631         PaxosInstance& pins_impl)
632     : pins_state_(pins_state)
633     , pins_impl_(pins_impl)
634 {
635     if (pins_impl.chosen()) {
636         assert(pins_impl.has_accepted_value());
637     }
638 }
639
640 std::tuple<int, bool, std::unique_ptr<Message>>
641     PInsWrapper::Step(const Message& msg)
642 {
643     assert(msg.index() == pins_impl_.index());
644         if (IsChosen()) {
645                 return stepChosen(msg);
646         }
647
648         assert(false == IsChosen());
649         return stepNotChosen(msg);
650 }
651
652 std::tuple<int, bool, std::unique_ptr<Message>>
653     PInsWrapper::stepChosen(const Message& msg)
654 {
655     assert(true == IsChosen());
656     bool write = false;
657     std::unique_ptr<Message> rsp_msg = nullptr;
658
```

```
659     assert(true == pins_impl_.has_promised_num());
660     assert(true == pins_impl_.has_accepted_num());
661     assert(true == pins_impl_.has_accepted_value());
662     switch (msg.type()) {
663     case MessageType::CHOSEN:
664                 // check
665                 if (msg.has_accepted_value())
666                 {
667             // TODO:
668                         if ((msg.accepted_value().data() !=
669                                         pins_impl_.accepted_value().data()) ||
670                                         (msg.accepted_value().reqid() !=
671                                          pins_impl_.accepted_value().reqid()))
672                         {
673                                 logerr("IMPORTANT INCONSISTENT index %" PRIu64
674                                                 " from %u to %u",
675                                                 msg.index(), msg.from(), msg.to());
676                 assert(false);
677                         }
678                 }
679     default:
680         break;
681
682         case MessageType::GET_CHOSEN:
683     case MessageType::PROP:
684         case MessageType::ACCPT:
685     case MessageType::FAST_ACCPT:
686         rsp_msg = cutils::make_unique<Message>();
687         assert(nullptr != rsp_msg);
688
689         rsp_msg->set_type(MessageType::CHOSEN);
690         rsp_msg->set_index(msg.index());
691         rsp_msg->set_key(msg.key());
692         rsp_msg->set_from(msg.to());
693         rsp_msg->set_to(msg.from());
694
695         rsp_msg->set_proposed_num(pins_impl_.proposed_num());
696         rsp_msg->set_promised_num(pins_impl_.promised_num());
697         rsp_msg->set_accepted_num(pins_impl_.accepted_num());
698                 rsp_msg->set_timestamp(time(NULL));
699         set_accepted_value(rsp_msg, pins_impl_.accepted_value());
700         break;
701     }
702
703     assert(false == write);
704     return std::make_tuple(0, write, move(rsp_msg));
705 }
706
```

```
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;

```

```
803     case MessageType::ACCPT:
804     case MessageType::FAST_ACCPT:
805         {
806             assert(msg.has_accepted_value());
807             bool fast_accept = MessageType::FAST_ACCPT == msg.type();
808             if (updateAccepted(
809                         msg.proposed_num(),
810                         msg.accepted_value(), fast_accept, pins_impl_)) {
811                 // accepted other
812                 write = true;
813             }
814 
815             rsp_msg_type = fast_accept
816                 ? MessageType::FAST_ACCPT_RSP : MessageType::ACCPT_RSP;
817         }
818         break;
819 
820     // proposer
821     case MessageType::PROP_RSP:
822     case MessageType::ACCPT_RSP:
823     case MessageType::FAST_ACCPT_RSP:
824         // start a propose
825     case MessageType::BEGIN_PROP:
826     case MessageType::TRY_PROP:
827     case MessageType::BEGIN_FAST_PROP:
828         case MessageType::TRY_REDO_PROP: // new add:
829         if (nullptr == pins_state_) {
830             logdebug("pins_state nullptr but recv msgtype %d",
831                     static_cast<int>(msg.type()));
832             break; // just ignore all proposer releated msg
833         }
834 
835         assert(nullptr != pins_state_);
836         assert(msg.key() == pins_state_->GetKey());
837                 assert(msg.index() == pins_state_->GetIndex());
838         std::tie(write, rsp_msg_type) = pins_state_->Step(msg, pins_impl_);
839         break;
840 
841     default:
842         assert(false);
843         break;
844     }
845 
846     auto rsp_msg = produceRsp(msg, rsp_msg_type);
847     if (MessageType::CHOSEN == rsp_msg_type) {
848         assert(nullptr != rsp_msg);
849         assert(false == IsChosen());
850         markChosen();
```

```
851     }
852     return std::make_tuple(0, write, std::move(rsp_msg));
853 }
854 
855 std::unique_ptr<Message>
856 PInsWrapper::produceRsp(const Message& msg, MessageType rsp_msg_type)
857 {
858     if (MessageType::NOOP == rsp_msg_type) {
859         return nullptr;
860     }
861 
862     std::unique_ptr<Message> rsp_msg = cutils::make_unique<Message>();
863     assert(nullptr != rsp_msg);
864     rsp_msg->set_key(msg.key());
865     rsp_msg->set_index(msg.index());
866     rsp_msg->set_from(msg.to());
867     rsp_msg->set_type(rsp_msg_type);
868     rsp_msg->set_proposed_num(msg.proposed_num());
869         rsp_msg->set_timestamp(time(NULL));
870     switch (rsp_msg_type) {
871 
872     // accepter
873     case MessageType::PROP_RSP:
874         rsp_msg->set_promised_num(pins_impl_.promised_num());
875         assert(rsp_msg->promised_num() >= rsp_msg->proposed_num());
876 
877                 // TODO: add test
878                 // promised or reject => both need send back accepted_num
879                 if (pins_impl_.has_accepted_num()) {
880                         assert(pins_impl_.has_accepted_value());
881                         rsp_msg->set_accepted_num(pins_impl_.accepted_num());
882                         set_accepted_value(rsp_msg, pins_impl_.accepted_value());
883                 }
884 
885         rsp_msg->set_to(msg.from());
886         break;
887 
888     case MessageType::ACCPT_RSP:
889     case MessageType::FAST_ACCPT_RSP:
890         {
891             assert(pins_impl_.has_promised_num());
892             // => reject ?
893             // assert(pins_impl_.has_accepted_num());
894             // assert(pins_impl_.has_accepted_value());
895 
896             // TODO: ? send back pins_impl_.promised_num as a hint
897             auto accepted_num =
898                 pins_impl_.has_accepted_num() ? pins_impl_.accepted_num() : 0;
```

```
899             rsp_msg->set_accepted_num(accepted_num);
900             if (accepted_num != msg.proposed_num()) {
901                 // reject => return promised_num as a hint
902                 if (0 == accepted_num) {
903                     rsp_msg->set_promised_num(pins_impl_.promised_num());
904                 }
905             }
906             rsp_msg->set_to(msg.from());
907         }
908         break;
909
910     // proposer
911     case MessageType::PROP:
912         assert(nullptr != pins_state_);
913         rsp_msg->set_proposed_num(pins_impl_.proposed_num());
914         // set_to 0 => broad-cast
915         rsp_msg->set_to(0);
916                 assert(0 < cutils::get_prop_cnt(rsp_msg->proposed_num()));
917         break;
918
919     case MessageType::ACCPT:
920     case MessageType::FAST_ACCPT:
921         assert(nullptr != pins_state_);
922         assert(pins_impl_.has_promised_num());
923         assert(pins_impl_.has_accepted_num());
924         assert(pins_impl_.has_accepted_value());
925
926                 assert(pins_impl_.proposed_num() == pins_state_->GetProposedNum());
927                 assert(pins_state_->HasProposingValue());
928                 assert(pins_impl_.accepted_value().data() ==
929                                 pins_state_->GetProposingValue().data());
930                 assert(pins_impl_.accepted_value().reqid() ==
931                                 pins_state_->GetProposingValue().reqid());
932                 rsp_msg->set_proposed_num(pins_state_->GetProposedNum());
933                 set_accepted_value(rsp_msg, pins_state_->GetProposingValue());
934         rsp_msg->set_to(0);
935
936                 // check
937                 if (MessageType::ACCPT == rsp_msg_type) {
938                         assert(0 < cutils::get_prop_cnt(rsp_msg->proposed_num()));
939                 }
940                 else {
941                         assert(MessageType::FAST_ACCPT == rsp_msg_type);
942                         assert(0 == cutils::get_prop_cnt(rsp_msg->proposed_num()));
943                 }
944         break;
945
946     case MessageType::CHOSEN:
```

```
947             assert(MessageType::CHOSEN != msg.type());
948             assert(pins_impl_.has_promised_num());
949             assert(pins_impl_.has_accepted_num());
950             assert(pins_impl_.has_accepted_value());
951             rsp_msg->set_proposed_num(pins_impl_.proposed_num());
952             set_accepted_value(rsp_msg, pins_impl_.accepted_value());
953             rsp_msg->set_to(0);
954             break;
955         default:
956             assert(false);
957             break;
958         }
959 
960         assert(rsp_msg->from() == msg.to());
961         return rsp_msg;
962     }
963 
964     } // namespace paxos
965 
```