


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

```
1
2  /*
3   * Tencent is pleased to support the open source community by making PaxosStore available.
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 file except in compliance with the License.
6   * https://opensource.org/licenses/BSD-3-Clause
7   * Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an
8   *
9   */
10
11
12 #include <cassert>
13 #include "plog_wrapper.h"
14 #include "paxos.pb.h"
15 #include "pins_wrapper.h"
16 #include "plog_helper.h"
17 #include "cutils/mem_utils.h"
18 #include "cutils/log_utils.h"
19 #include "cutils/hassert.h"
20 #include "cutils/id_utils.h"
21
22 namespace {
23
24 std::unique_ptr<paxos::Message>
25 buildNoopRspMsg(const paxos::Message& req_msg, const uint64_t max_index)
26 {
27     auto rsp_msg = cutils::make_unique<paxos::Message>();
28     assert(nullptr != rsp_msg);
29
30     rsp_msg->set_type(paxos::MessageType::NOOP);
31     rsp_msg->set_from(req_msg.to());
32     rsp_msg->set_to(req_msg.from());
33     rsp_msg->set_key(req_msg.key());
34     rsp_msg->set_index(max_index);
35     rsp_msg->set_proposed_num(0);
36     return rsp_msg;
37 }
38
39 bool belong_to(uint64_t reqid, uint16_t member_id)
40 {
41     if (0 == reqid) {
42         return false;
43     }
44
45     uint16_t req_member_id = 0;
46     uint16_t req_cnt = 0;
47     std::tie(req_member_id, req_cnt) =
48         cutils::IDGenerator::decompose(reqid);
49     return req_member_id == member_id;
50 }
51
52 } // namespace
```

```

53
54 namespace paxos {
55
56
57 PLogWrapper::PLogWrapper(
58     uint8_t selfid,
59     uint16_t member_id,
60     const std::string& key,
61     PInsAliveState* pins_state,
62     PaxosLog& plog_impl)
63 : selfid_(selfid)
64 , member_id_(member_id)
65 , key_(key)
66 , pins_state_(pins_state)
67 , plog_impl_(plog_impl)
68 {
69     assert(0 < selfid_);
70     assert(is_slim(plog_impl_));
71
72     if (0 < plog_impl_.entries_size()) {
73         assert(2 >= plog_impl_.entries_size());
74         auto min_index = get_min_index(plog_impl_);
75         auto max_index = get_max_index(plog_impl_);
76         assert(min_index == max_index || min_index + 1 == max_index);
77
78         uint64_t index_sofar = 0;
79         for (int idx = 0; idx < plog_impl_.entries_size(); ++idx) {
80             const auto& ins = plog_impl_.entries(idx);
81             if (ins.chosen()) {
82                 assert(ins.has_promised_num());
83                 assert(ins.has_accepted_num());
84                 assert(ins.has_accepted_value());
85             }
86
87             assert(0 == ins.index() || index_sofar < ins.index());
88             index_sofar = ins.index();
89         }
90     }
91 }
92
93 PLogWrapper::~PLogWrapper() = default;
94
95 std::tuple<int, std::unique_ptr<PInsWrapper>>
96 PLogWrapper::getInstance(const uint64_t msg_index)
97 {
98     assert(0 < msg_index);
99     auto min_index = get_min_index(plog_impl_);
100     auto max_index = get_max_index(plog_impl_);
101     assert(min_index <= max_index);
102
103     if (msg_index != min_index && msg_index != max_index) {
104         if (msg_index < max_index) {
105             return std::make_tuple(-1, nullptr);
106         }
107
108         assert(msg_index > max_index);
109         // create a new pending ins
110         // => drop prev ins
111         if (nullptr != pins_state_) {
112             pins_state_->SendNotify();
113             pins_state_ = nullptr;
114         }
115
116         assert(nullptr == pins_state_);
117
118         auto ins = plog_impl_.add_entries();

```

```

119     assert(nullptr != ins);
120     ins->set_index(msg_index);
121 }
122
123 paxos::PaxosInstance* ins = nullptr;
124 for (int idx = 0; idx < plog_impl_.entries_size(); ++idx) {
125     assert(nullptr != plog_impl_.mutable_entries(idx));
126     if (msg_index != plog_impl_.entries(idx).index()) {
127         continue;
128     }
129
130     ins = plog_impl_.mutable_entries(idx);
131     assert(nullptr != ins);
132     break;
133 }
134
135 assert(nullptr != ins);
136 assert(msg_index == ins->index());
137 return std::make_tuple(
138     0, cutils::make_unique<PInsWrapper>(pins_state_, *ins));
139 }
140
141 std::tuple<int, std::unique_ptr<Message>>
142 PLogWrapper::stepInvalidIndex(const Message& msg)
143 {
144     assert(is_slim(plog_impl_));
145     const int entries_size = plog_impl_.entries_size();
146     if (0 == entries_size) {
147         return std::make_tuple(0, nullptr);
148     }
149
150     auto max_index = get_max_index(plog_impl_);
151     assert(0 < max_index && msg.index() < max_index);
152
153     std::unique_ptr<Message> rsp_msg;
154     int err = 0;
155     std::unique_ptr<PInsWrapper> pins = nullptr;
156     switch (msg.type())
157     {
158     case MessageType::GET_CHOSEN:
159     case MessageType::PROP:
160     case MessageType::ACCPY:
161     case MessageType::FAST_ACCPY:
162     {
163         auto chosen_ins = get_chosen_ins(plog_impl_);
164         if (nullptr == chosen_ins) {
165             break; // do nothing
166         }
167
168         assert(nullptr != chosen_ins);
169         assert(msg.index() < chosen_ins->index());
170
171         Message fake_msg = msg;
172         fake_msg.set_type(MessageType::GET_CHOSEN);
173         fake_msg.set_index(chosen_ins->index());
174
175         std::tie(err, pins) = getInstance(fake_msg.index());
176         assert(0 == err);
177         assert(nullptr != pins);
178         bool write = false;
179         // pins => chosen_ins !!!
180         std::tie(err, write, rsp_msg) = pins->Step(fake_msg);
181         assert(0 == err);
182         assert(false == write);
183         assert(nullptr != rsp_msg);
184         assert(MessageType::CHOSEN == rsp_msg->type());

```

```

185         assert(chosen_ins->index() == rsp_msg->index());
186         rsp_msg->set_to(msg.from());
187     }
188     break;
189 case MessageType::CHOSEN:
190     {
191         if (msg.index() + 1 != max_index ||
192             plog_impl_.entries(entries_size-1).chosen()) {
193             break; // do nothing;
194         }
195
196         assert(msg.index() + 1 == max_index);
197         assert(false == plog_impl_.entries(entries_size-1).chosen());
198         assert(1 == entries_size);
199
200         PaxosInstance new_ins;
201         new_ins.set_index(msg.index());
202         pins = cutils::make_unique<PInsWrapper>(nullptr, new_ins);
203         assert(nullptr != pins);
204
205         PaxosLog plog_new;
206         {
207             auto add_ins = plog_new.add_entries();
208             assert(nullptr != add_ins);
209             add_ins->Swap(&new_ins);
210
211             add_ins = plog_new.add_entries();
212             assert(nullptr != add_ins);
213             add_ins->Swap(plog_impl_.mutable_entries(entries_size-1));
214         }
215
216         plog_new.Swap(&plog_impl_);
217         assert(2 == plog_impl_.entries_size());
218         assert(is_slim(plog_impl_));
219         setDiskWrite();
220         setUpdateChosen();
221     }
222     break;
223 default:
224     break;
225 }
226
227 return std::make_tuple(0, std::move(rsp_msg));
228 }
229
230 std::tuple<int, std::unique_ptr<Message>>
231 PLogWrapper::Step(const Message& msg)
232 {
233     if ((0 == msg.index()) ||
234         key_ != msg.key() ||
235         static_cast<uint32_t>(selfid_) != msg.to()) {
236
237         // GET_CHOSEN: fix case;
238         if (0 == msg.index() && MessageType::GET_CHOSEN == msg.type()) {
239             return stepInvalidIndex(msg);
240         }
241
242         logerr("msg.index %" PRIu64 " selfid %d msg.to %u",
243              msg.index(),
244              static_cast<int>(selfid_), msg.to());
245         return std::make_tuple(-1, nullptr);
246     }
247
248     std::unique_ptr<Message> rsp_msg = nullptr;
249     {
250         bool write = false;

```

```

251     int err = 0;
252     std::unique_ptr<PInsWrapper> pins = nullptr;
253     // may update committed_index;
254     std::tie(err, pins) = getInstance(msg.index());
255     if (0 != err) {
256         assert(nullptr == pins);
257         assert(-1 == err);
258         // msg_index < std::max(chosen_index, pending_index);
259         return stepInvalidIndex(msg);
260     }
261
262     assert(0 == err);
263     assert(nullptr != pins);
264     const bool already_chosen = pins->IsChosen();
265     // - rsp msg;
266     // - chosen ?
267     // => chosen_ins = pending_ins; pending_ins.clear();
268     std::tie(err, write, rsp_msg) = pins->Step(msg);
269     if (0 != err) {
270         assert(false == write);
271         assert(nullptr == rsp_msg);
272         return std::make_tuple(err, nullptr);
273     }
274
275     const bool now_chosen = pins->IsChosen();
276     pins = nullptr;
277     /*
278         logdebug("key %" PRIu64 " %" PRIu64 " already_chosen %d now_chosen %d"
279                 " reqmsgtype %d rsp_msg %p rsp_msg_type %d",
280                 msg.logid(), msg.index(), already_chosen, now_chosen,
281                 static_cast<int>(msg.type()),
282                 rsp_msg.get(), nullptr == rsp_msg ? -1 : static_cast<int>(rsp_msg->type()));
283     */
284     if (false == already_chosen && now_chosen) {
285         setDiskWrite();
286     }
287
288     assert(nullptr == pins);
289     if (write) {
290         setDiskWrite();
291     }
292
293     auto do_shrink = shrink_plog(plog_impl_);
294     if (1 == do_shrink) {
295         setDiskWrite();
296     }
297 }
298
299 if (nullptr != rsp_msg) {
300     assert(rsp_msg->index() == msg.index());
301     assert(rsp_msg->key() == msg.key());
302     assert(rsp_msg->from() == msg.to());
303     assert(rsp_msg->from() == static_cast<uint32_t>(selfid_));
304     assert(rsp_msg->key() == key_);
305 }
306
307 return std::make_tuple(0, std::move(rsp_msg));
308 }
309
310 std::tuple<
311     int,
312     std::shared_ptr<PInsAliveState>,
313     std::unique_ptr<Message>>
314 PLogWrapper::Set(
315     uint64_t reqid,
316     const std::string& raw_value,

```

```

317         bool do_fast_acpt)
318     {
319         if (nullptr != pins_state_) {
320             return std::make_tuple(-10, nullptr, nullptr);
321         }
322
323         assert(nullptr == pins_state_);
324         auto max_ins = get_max_ins(plog_impl_);
325         if (nullptr != max_ins && false == max_ins->chosen()) {
326             return PreemptSet(reqid, raw_value);
327         }
328
329         assert(nullptr == max_ins || max_ins->chosen());
330         return NormalSet(reqid, raw_value, do_fast_acpt);
331     }
332
333     std::tuple<
334         int,
335         std::shared_ptr<PInsAliveState>,
336         std::unique_ptr<Message>>
337     PLogWrapper::NormalSet(
338         uint64_t reqid,
339         const std::string& data, const bool do_fast_acpt)
340     {
341         assert(is_slim(plog_impl_));
342         if (nullptr != pins_state_) {
343             return std::make_tuple(-10, nullptr, nullptr);
344         }
345
346         assert(nullptr == pins_state_);
347         auto max_ins = get_max_ins(plog_impl_);
348         if (nullptr != max_ins && false == max_ins->chosen()) {
349             return std::make_tuple(ErrorCode::BUSY, nullptr, nullptr);
350         }
351
352         assert(nullptr == max_ins || max_ins->chosen());
353         uint64_t propose_index =
354             nullptr == max_ins ? 1 : max_ins->index() + 1;
355
356         Message msg;
357         msg.set_type(MessageType::BEGIN_PROP);
358         msg.set_from(selfid_);
359         msg.set_to(selfid_);
360         msg.set_key(key_);
361         msg.set_index(propose_index);
362         {
363             auto entry = msg.mutable_accepted_value();
364             assert(nullptr != entry);
365             entry->set_reqid(reqid);
366             entry->set_data(data);
367         }
368
369         // must be the case
370         // assert(propose_index == pins_state_->GetIndex());
371         // assert(PropState::NIL == pins_state_->GetPropState());
372         bool can_do_fast = false;
373         if (nullptr != max_ins) {
374             assert(max_ins->chosen());
375             if (belong_to(max_ins->accepted_value().reqid(), member_id_)) {
376                 can_do_fast = true;
377             }
378         }
379
380         if (do_fast_acpt && can_do_fast) {
381             msg.set_type(MessageType::BEGIN_FAST_PROP);
382         }

```

```

383
384     msg.set_proposed_num(
385         cutils::prop_num_compose(selfid_, 0));
386
387     auto shared_pins_state =
388         std::make_shared<PinsAliveState>(
389             key_, propose_index, msg.proposed_num());
390     assert(nullptr != shared_pins_state);
391     pins_state_ = shared_pins_state.get();
392     // assert(0 == cutils::get_prop_cnt(pins_state_>GetProposedNum()));
393
394     auto new_ins = plog_impl_.add_entries();
395     assert(nullptr != new_ins);
396     new_ins->set_index(propose_index);
397
398     int ret = 0;
399     std::unique_ptr<Message> rsp_msg;
400     std::tie(ret, rsp_msg) = Step(msg);
401     if (0 != ret) {
402         pins_state_ = nullptr;
403         return std::make_tuple(ret, nullptr, nullptr);
404     }
405
406     assert(0 == ret);
407     assert(nullptr != rsp_msg);
408     assert(shared_pins_state->GetIndex() == rsp_msg->index());
409     return std::make_tuple(
410         0, std::move(shared_pins_state), std::move(rsp_msg));
411 }
412
413 std::tuple<
414     int,
415     std::shared_ptr<PinsAliveState>,
416     std::unique_ptr<Message>>
417 PLogWrapper::PreemptSet(uint64_t reqid, const std::string& data)
418 {
419     assert(is_slim(plog_impl_));
420     if (nullptr != pins_state_) {
421         return std::make_tuple(-10, nullptr, nullptr);
422     }
423
424     // must be
425     assert(nullptr == pins_state_);
426     auto max_ins = get_max_ins(plog_impl_);
427     auto chosen_ins = get_chosen_ins(plog_impl_);
428     if (nullptr == max_ins ||
429         nullptr == chosen_ins ||
430         max_ins->index() != chosen_ins->index() + 1) {
431         if (!(nullptr != max_ins && 1 == max_ins->index())) {
432             return std::make_tuple(-11, nullptr, nullptr);
433         }
434     }
435
436     assert(nullptr != max_ins);
437     assert(false == max_ins->chosen());
438     uint64_t propose_index = max_ins->index();
439     assert(0 < propose_index);
440
441     Message msg;
442     msg.set_type(MessageType::TRY_PROP);
443     msg.set_from(selfid_);
444     msg.set_to(selfid_);
445     msg.set_key(key_);
446     msg.set_index(propose_index);
447     msg.set_proposed_num(
448         cutils::PropNumGen(

```

```

449         selfid_, 0).Next(max_ins->proposed_num());
450     hassert(msg.proposed_num() > max_ins->proposed_num(),
451             "msg.proposed_num %" PRIu64
452             " max_ins.proposed_num %" PRIu64,
453             msg.proposed_num(), max_ins->proposed_num());
454     {
455         auto entry = msg.mutable_accepted_value();
456         assert(nullptr != entry);
457         entry->set_reqid(reqid);
458         entry->set_data(data);
459     }
460
461     auto shared_pins_state =
462         std::make_shared<PinsAliveState>(
463             key_, propose_index, msg.proposed_num());
464     assert(nullptr != shared_pins_state);
465     pins_state_ = shared_pins_state.get();
466
467     int ret = 0;
468     std::unique_ptr<Message> rsp_msg;
469     std::tie(ret, rsp_msg) = Step(msg);
470     if (0 != ret) {
471         pins_state_ = nullptr;
472         return std::make_tuple(ret, nullptr, nullptr);
473     }
474
475     assert(0 == ret);
476     assert(nullptr != rsp_msg);
477     assert(shared_pins_state->GetIndex() == rsp_msg->index());
478     return std::make_tuple(
479         0, std::move(shared_pins_state), std::move(rsp_msg));
480 }
481
482 std::tuple<int, std::unique_ptr<Message>>
483 PLogWrapper::TryRedoProp()
484 {
485     assert(is_slim(plog_impl_));
486     if (nullptr == pins_state_) {
487         return std::make_tuple(-20, nullptr);
488     }
489
490     assert(nullptr != pins_state_);
491     auto max_ins = get_max_ins(plog_impl_);
492     if (nullptr == max_ins || max_ins->chosen()) {
493         return std::make_tuple(1, nullptr);
494     }
495
496     assert(nullptr != max_ins && false == max_ins->chosen());
497     assert(pins_state_->GetIndex() == max_ins->index());
498
499     Message msg;
500     msg.set_type(MessageType::TRY_REDO_PROP);
501     msg.set_from(selfid_);
502     msg.set_to(selfid_);
503     msg.set_key(key_);
504     msg.set_index(max_ins->index());
505     msg.set_proposed_num(
506         cutils::PropNumGen(selfid_, 0).Next(max_ins->proposed_num()));
507     hassert(msg.proposed_num() > max_ins->proposed_num(),
508             "msg.proposed_num %" PRIu64
509             " max_ins->proposed_num %" PRIu64,
510             msg.proposed_num(), max_ins->proposed_num());
511
512     auto entry = msg.mutable_accepted_value();
513     assert(nullptr != entry);
514     // case 1:

```



```
515     if (pins_state_->HasProposingValue()) {
516         entry->set_reqid(pins_state_->GetProposingValue().reqid());
517         entry->set_data(pins_state_->GetProposingValue().data());
518         return Step(msg);
519     }
520
521     assert(false == pins_state_->HasProposingValue());
522     // case 2:
523     if (max_ins->has_accepted_value()) {
524         entry->set_reqid(max_ins->accepted_value().reqid());
525         entry->set_data(max_ins->accepted_value().data());
526         return Step(msg);
527     }
528
529     assert(false == max_ins->has_accepted_value());
530     // case 3:
531     auto chosen_ins = get_chosen_ins(plog_impl_);
532     if (nullptr == chosen_ins ||
533         chosen_ins->index() + 1 != max_ins->index()) {
534         logerr("FAILED LOCAL OUT: chosen_ins.index %" PRIu64
535              " pending_ins.index %" PRIu64,
536              chosen_ins->index(), max_ins->index());
537         return std::make_tuple(-21, nullptr);
538     }
539
540     assert(nullptr != chosen_ins &&
541            chosen_ins->index() + 1 == max_ins->index());
542     entry->set_reqid(0);
543     entry->set_data(chosen_ins->accepted_value().data());
544     return Step(msg);
545 }
546
547 PInsAliveState*
548 PLogWrapper::SetPInsAliveState(PInsAliveState* new_pins_state)
549 {
550     std::swap(pins_state_, new_pins_state);
551     return new_pins_state;
552 }
553
554
555 } // namespace paxos
556
557
```