```
 1 #ifndef MSNetLoopRunnableHEADER
 2 #define MSNetLoopRunnableHEADER
 3
 4 #include <MSNet/MSNetLoop.H>
 5 #include <string>
 6 #include <vector>
 7 #include <memory>
 8
 9
10 namespace fusionbase {
11
12     typedef std::shared_ptr<MSNetLoop>     MSNetLoopPtr;
13
14     class MSNetLoopRunnable
15     {
16     public:
17         void set(MSNetLoopPtr netLoopPtr_)
18         {
19             if(!netLoopPtr_)
20             {
21                 return;
22             }
23             _netLoopPtr = netLoopPtr_;
24
25             if(!start() )
26             {
27                 return;
28             }
29             _isSet =true;
30         }
31
32         virtual ~MSNetLoopRunnable(){}
33         virtual std::string getId() const{ return "";}
34         virtual bool start() = 0;
35
36         void run()
37         {
38             if(!_isSet || (_netLoopPtr->isLooping()) )//
   should not happened
39             {
40                 return ;
41             }
42             _netLoopPtr->loop();
43         }
44
```

```
45      protected:
46          MSNetLoopPtr          _netLoopPtr;
47      private:
48          bool                  _isSet{false};
49      };
50
51      typedef std::shared_ptr<MSNetLoopRunnable>
    MSNetLoopRunnablePtr;
52      typedef std::vector<MSNetLoopRunnablePtr>
    MSNetLoopRunnables;
53
54  }// namespace fusionbase
55
56
57
58  #endif // MSNetLoopRunnableHEADER
59
60
61  #ifndef MSNetLoopsRunnerHEADER
62  #define MSNetLoopsRunnerHEADER
63
64  #include <fusionbase/MSNetLoopRunnable.H>
65  #include <MSLog/MSLog.H>
66  #include <future>
67  #include <tuple>
68
69  /*
70   * Typical usage
71   *
72  // running from from 1 caller ( main) thread, 1
    runnableImpl
73   void test0(){
74
75    fusionbase::MSNetLoopRunner runnables;
76    runnables.add(std::make_shared<RunnableImpl1>());
77    runnables.run(std::launch::deferred);
78  }
79
80   // running from from 1 caller ( main) thread, 2
    runnableImpls
81   void test1(){
82
83    fusionbase::MSNetLoopRunner runnables;
84    runnables.add(std::make_shared<RunnableImpl1>());
85    runnables.add(std::make_shared<RunnableImpl2>());
```

```
 86    runnables.run(std::launch::deferred);
 87 }
 88
 89 // running from from 1 child thread, 2 runnableImpls
 90 void test1()
 91 {
 92    fusionbase::MSNetLoopRunner runnables;
 93    runnables.add(std::make_shared<RunnableImpl1>());
 94    runnables.add(std::make_shared<RunnableImpl2>());
 95    runnables.run();
 96 }
 97
 98 // running from from 2 child thread
 99 // 2 Runnables in the first thread and and 1 Runnable in
    the second thread
100 void test3(){
101
102 fusionbase::MSNetLoopsRunner runnables(2);
103    runnables.add(std::make_shared<RunnableImpl1>(), 0);
104    runnables.add(std::make_shared<RunnableImpl2>(), 0);
105    runnables.add(std::make_shared<RunnableImpl3>(), 1);
106    runnables.run();
107
108 }
109
110  */
111
112 namespace fusionbase {
113 /*
114  * one MSNetLoop should be run in one thread.
115  */
116
117     enum class ExceptionPolicy { THROW_ALL, THROW_NONE};
118
119 /*
120  * Runner for one MSNetLoop - one thread.
121  * launchMode_ == std::launch::deferred will run in the
    caller thread.
122  * launchMode_ == std::launch::async will run in a child
    thread.
123  * Caller thread is blocked on the run().
124  * All exception caught in the caller thread and rethrown
     if exceptionPolicy_ = ExceptionPolicy::THROW_ALL
125  */
126     class MSNetLoopRunner
```

```
127     {
128     public:
129
130         void add(MSNetLoopRunnablePtr runnable_ )
131         {
132             _runnables.emplace_back(std::move(runnable_))
    ;
133         }
134
135         void run(std::launch launchMode_ = std::launch::
    async, ExceptionPolicy exceptionPolicy_ = ExceptionPolicy
    ::THROW_ALL )
136         {
137             _future = std::async(launchMode_, [this]{
    syncRun();});
138             try
139             {
140                 _future.wait();
141             }
142             catch (const std::exception& exception_)
143             {
144                 MSlogError <<"Exception "<<exception_.
    what()<< send;
145                 if(exceptionPolicy_ == ExceptionPolicy::
    THROW_ALL)
146                 {
147                     throw;
148                 }
149             }
150             catch (...)
151             {
152                 MSlogError <<"Unknown exception "<< send;
153                 if(exceptionPolicy_ == ExceptionPolicy::
    THROW_ALL)
154                 {
155                     throw;
156                 }
157             }
158         }
159
160     private:
161         void syncRun()
162         {
163             _netLoopPtr = std::make_shared<MSNetLoop>();
164             for( MSNetLoopRunnablePtr& runnable:
```

```
164 _runnables)
165             {
166                 runnable->set(_netLoopPtr);
167             }
168             if(!_runnables.empty())
169             {
170                 _runnables.at(0)->run();
171             }
172         }
173     private:
174         std::future<void>                   _future;
175         MSNetLoopPtr                        _netLoopPtr;
176         MSNetLoopRunnables                  _runnables;
177     };
178
179 /*
180  * Runner for multiple loops. Each loop in its own child
    thread.
181  * launchMode_ == std::launch::deferred is forbidden here
    .
182  * Caller thread is blocked  on the run().
183  * All exception caught in the caller thread and rethrown
     if exceptionPolicy_ = ExceptionPolicy::THROW_ALL
184  */
185     class MSNetLoopsRunner
186     {
187         typedef std::tuple< std::future<void>,
    MSNetLoopPtr, MSNetLoopRunnables> Runner;
188         typedef std::vector<Runner> Runners;
189     public:
190         explicit MSNetLoopsRunner(size_t threadNum):
191                 _runners(threadNum)
192         {
193         }
194
195         void add(MSNetLoopRunnablePtr runnable_, size_t
    pos  )
196         {
197             Runner& runner = _runners.at(pos);
198             MSNetLoopRunnables& runnables  = std::get<
    MSNetLoopRunnables>(runner);
199             runnables.emplace_back(runnable_);
200         }
201
202         /*
```

```
203         * std::launch::deferred should be forbidden here
     because more than one runnable
204         * with std::launch::deferred will cause dealock
    on first wait.
205         * MSNetLoopRunner should be used with
    launchMode_ == std::launch::deferred
206         */
207        void run(ExceptionPolicy exceptionPolicy_ =
    ExceptionPolicy::THROW_NONE )
208        {
209            for(Runner& runner: _runners)
210            {
211                Runner* runnerPtr = &runner;
212                std::future<void>& f = std::get<std::
    future<void> >(runner);
213                f = std::async(std::launch::async, [this,
     runnerPtr]{syncRun(runnerPtr);});// run all with std::
    launch::async
214            }
215
216            for(Runner& runner: _runners)
217            {
218                std::future<void>& f = std::get<std::
    future<void> >(runner);
219                try
220                {
221                    f.wait();
222                }
223                catch (const std::exception& exception_)
224                {
225                    MSlogError <<"Exception "<<exception_
    .what()<< send;
226                    if(exceptionPolicy_ ==
    ExceptionPolicy::THROW_ALL)
227                    {
228                        throw;
229                    }
230                }
231                catch (...)
232                {
233                    MSlogError <<"Unknown exception "<<
    send;
234                    if(exceptionPolicy_ ==
    ExceptionPolicy::THROW_ALL)
235                    {
```

```
236                         throw;
237                     }
238                 }
239             }
240         }
241
242     private:
243         void syncRun(Runner* runner)
244         {
245             MSNetLoopPtr& loopPtr = std::get<MSNetLoopPtr
    >(*runner);
246             loopPtr = std::make_shared<MSNetLoop>();
247             MSNetLoopRunnables& runnables = std::get<
    MSNetLoopRunnables>(*runner);
248
249             for( MSNetLoopRunnablePtr& runnable:
    runnables)
250             {
251                 runnable->set(loopPtr);
252             }
253
254             if(!runnables.empty())
255             {
256                 runnables.at(0)->run();
257             }
258         }
259
260     private:
261         Runners _runners;
262     };
263
264 }// namespace fusionbase
265
266
267
268 #endif // MSNetLoopsRunnerHEADER
269
270 #ifndef CpsMessageSubscriberHEADER
271 #define CpsMessageSubscriberHEADER
272
273
274 #include <fusionbase/ControlTimestamp.H>
275 #include <fusionbase/MSNetLoopRunnable.H>
276 #include <fusionbase/MSNetLoopsRunner.H>
277 #include <CPS/CPSClassicSubscriber.H>
```

```
278 #include <CPS/CPSMessage.h>
279 #include <boost/thread/executors/basic_thread_pool.hpp>
280 #include <memory>
281
282
283 namespace fusionbase {
284
285     class CpsMessageSubscriber : public MSNetLoopRunnable
286     {
287     public:
288         CpsMessageSubscriber() = default;// one thread -
    one loop - one subscriber
289
290         virtual ~CpsMessageSubscriber(){}
291
292         virtual bool start();
293         std::string getId() const override
294         {
295             return _baseConfig._name;
296         }
297     protected:
298         typedef std::unique_ptr<CPSClassicSubscriber>
    SubscriberPtr;
299
300         virtual void setConfig() = 0;
301         virtual void publicationCallback(const CPSMessage
     & message_) = 0;
302
303         virtual void init();
304         virtual void connectCallback();
305         virtual void disconnectCallback();
306         virtual void retryCallback(const MSNetID & id);
307         virtual void errorCallback(const std::string&
    error_, MSNetTCPConnectionBase & conn_);
308         virtual void bindSubscriptionCallback(const
    CPSMessage & message_);
309         virtual void setDeltaFilter(CPSSubscription&
    subscription_);
310     protected:
311         struct BaseConfig
312         {
313             std::string
    _name;
314             std::string
    _addr;
```

```
315             std::string
    _topic;
316             std::string
    _filter;
317             std::string
    _deltaFilter;
318             bool

    _kerberos{true};
319             int
    _maxConnectRetries{5};
320         };
321     protected:
322         BaseConfig
    _baseConfig;
323         SubscriberPtr                _subscriberPtr;
324     };
325
326     class SubscriberWithCatchup : public virtual
    CpsMessageSubscriber
327     {
328         typedef CpsMessageSubscriber Base;
329     public:
330         SubscriberWithCatchup() = default;
331
332         virtual ~SubscriberWithCatchup(){}
333         bool start()override;
334     protected:
335         virtual void catchup( const TimePointMs&
    catchupStart_) = 0;
336
337         void init() override;
338         void disconnectCallback() override;
339         void bindSubscriptionCallback(const CPSMessage &
    message_) override;
340
341         virtual void catchup();
342         virtual void startTimer();
343         virtual void stopTimer();
344         virtual TimePointMs calculateCatchupStart();
345         virtual void onLastNotificationTimer(const
    MSNetTimer&);
346     protected:
347         struct CatchupConfig
348         {
```

```
349             bool
    _catchupEnabled{true};// temp disable catchup in config
350             std::string
    _timestampVariablePrefix;
351             unsigned int
                                        _checkPoint{60};//sec
352             unsigned long                   _timerInterval{
    60};//sec
353             unsigned long                   _backInterval{5
    };//in _timerIntervals
354             unsigned long
    _backIntervalOnFirstStart{24};//sec in  hours
355         };
356     protected:
357         CatchupConfig
    _catchupConfig;
358         ControlTimestamp
    _timestamp;
359         MSNetTimer                          _timer;
360     };
361
362
363 // batch per message
364     template<typename T> class BatchSubscriber : public
    virtual CpsMessageSubscriber
365     {
366     public:
367         virtual ~BatchSubscriber(){}
368     protected:
369         typedef std::vector<T>   Batch;
370         virtual Batch makeBatch(const CPSMessage &
    message_) = 0;
371         virtual std::function<void()> makeTask(Batch
    batch_ ) = 0;
372         void submit( std::function<void()> task_ )
373         {
374             m_pool.submit(task_);
375         }
376         void publicationCallback(const CPSMessage&
    message_) override
377         {
378             Batch batch = makeBatch(message_);
379             std::function<void()> task = makeTask(std::
```

```
379 move(batch));
380             submit(std::move(task));
381         }
382     private:
383         boost::basic_thread_pool    m_pool{1};
384     };
385
386 //Limits on accumulation batch size and time
387 // could corresponds several messages
388     template<typename T> class BatchSubscriberWithLimits
    : public BatchSubscriber<T>
389     {   typedef typename BatchSubscriber<T>::Batch Batch;
390     public:
391         virtual ~BatchSubscriberWithLimits(){}
392     protected:
393         void publicationCallback(const CPSMessage &
    message_) override
394         {
395             try
396             {
397                 if(!_timer.isValid() )// on first call
    only
398                 {
399                     _timer = this->_netLoopPtr->
    callbackAfterDelay(_batchConfig._timerDelay,
400
              [this] () { onTimer(); } );
401                 }
402
403                 Batch batch = this->makeBatch(message_);
404                 _batch.insert(batch.begin(), batch.end()
    );
405                 if( (_batch.size() > _batchConfig.
    _batchSize) || _timerExpired)
406                 {
407                     std::function<void()> task = makeTask
    (std::move(batch));
408                     this->submit(std::move(task));
409                     _timerExpired =false;
410                     _timer.expirationInterval(
    _batchConfig._timerDelay);
411                 }
412             }
413             catch (const std::exception & e)
414             {
```

```
415                 MSlogError << "An exception occured " <<e
    .what() << send;
416             }
417         }
418
419     protected:
420         struct BatchConfig
421         {
422             unsigned long       _batchSize{1};
423             unsigned long       _timerDelay{std::
    numeric_limits<unsigned long >::max()};
424         };
425     private:
426         void onTimer()
427         {
428             _timerExpired =true;
429         }
430
431         BatchConfig                         _batchConfig;
432     private:
433         MSNetTimer                  _timer;
434         Batch
    _batch;
435         bool
    _timerExpired{false};
436     };
437
438 };//namespace fusionbase
439
440 #endif // CpsMessageProcessorBaseHEADER
441
```