```
#ifndef THREE_TIER_HEAP_EVENT_QUEUE_H
#define THREE_TIER_HEAP_EVENT_QUEUE_H

#include <vector>
#include <stack>
#include <algorithm>
#include "Avg.h"
#include "EventQueue.h"
#include "TwoTierHeapOfVectorsEventQueue.h"

BEGIN_NAMESPACE(xxxx)

/** Class to encapsulate information for Tier2 entries/buckets.

        This is a simple class that encapsulates a list of events (in
        eventList) all with exactly the same receive time to a given
        agent.  These objects are cached/reused by the scheduler queue to
        reduce memory allocation operations, particularly for the
        eventList because memory management turns out to be he most
        expensive operation.
*/
class HOETier2Entry {
private:
    /** The receive time of events in this tier2 entry. Note that all
        the events in this entry are must/are concurrent -- that is,
        destined for the same agent at the same time.
     */
    Time recvTime;

    /** The list of entities in this HOE entry class */
    std::vector<xxxx::Event*> eventList;

public:
    /** Constructor to create a tier2 entry with 1 initial event in
        it.

        \param[in] event The event to be added to this tier2 entry.
        The receive time of the event is used as the receive time
        value.
    */
    HOETier2Entry(xxxx::Event* event) : recvTime(event->getReceiveTime()),
                                        eventList(1, event) {}

    /** Reset the information in this tier2 entry.

        This method is synonymous to the constructor except, it is
        used to reset/recycle an existing tier2 entry.

        \param[in] event The event to be added to this tier2 entry.
        The receive time of the event is used as the receive time
        value.
    */
    void reset(xxxx::Event* event) {
        recvTime = event->getReceiveTime();
        eventList.clear();
        eventList.emplace_back(event);
    }

    /** Appends events to the EventContainer list.
     *
     *  The method is used to append concurrent events to their respective
     *  position in the tier2 container.
     */
    void updateContainer(xxxx::Event* event){
        eventList.emplace_back(event);
    }

    /** Obtain pointer to the first event in this list.

        \return Pointer to the first event in this list. This return
        value cannot/should-not be NULL.
    */
    inline xxxx::Event* getEvent() const {
        return eventList.front();
    }
```



```

    /** \brief compares the receive times of events

        The method is used to determine whether or not an event
        already exists in the tier2 container.

        \returns True if lhs receiveTime is equal to rhs receiveTime
    */
    inline bool operator==(const HOETier2Entry &rhs) {
        return (this->recvTime == rhs.recvTime);
    }

    inline bool operator<(const HOETier2Entry &rhs) {
        return (this->recvTime < rhs.recvTime);
    }

    inline Time getReceiveTime() const {
        return recvTime;
    }

    inline const std::vector<xxxx::Event*>& getEventList() const {
        return eventList;
    }

    inline std::vector<xxxx::Event*>& getEventList() {
        return eventList;
    }
};

/** A three-tier-heap aka "3tHeap" or "heap-of-heap" event queue for
    managing events.

    <p>This class provides a heap-of-heap based event queue for
    managing events for simulation.  The two-tiers are organized as
    follows:</p>

    <p><u>First tier:</u> This class uses standard C++ algorithms
    (such as: \c std::make_heap, \c std::push_heap, \c std::pop_heap)
    to manage a heap of events for each agent.  The events are stored
    in a backing std::vector in each agent.  It is the same per-agent
    infrastructure as used by Fibonacci heap (implemented in AgentPQ
    class).</p>

    <p><u>Second tier:</u> This class specifically handles the
    necessary behavior of the second tier of operations -- that is
    scheduling of agents by maintaining heap of agents.</p>

    \note On the long run it would be better to avoid reliance on
    std::push_heap or std::pop_heap methods due to implicit dependence
    in the fixHeap() method.
*/
class ThreeTierHeapEventQueue : public EventQueue {
public:
    /** The constructor for the TwoTierHeapEventQueue.

        The default (and only) constructor for this class.  The
        constructor does not have any specific task to perform other
        than set a suitable identifier in the base class.
    */
    ThreeTierHeapEventQueue();

    /** The destructor.

        The destructor does not have any special tasks to perform.
        All the dynamic memory management is handled by the standard
        containers (namely std::vector) that is used internally by
        this class.
    */
    ~ThreeTierHeapEventQueue();

    /** Add/register an agent with the event queue.

        <p>This method implements the corresponding API method in the
        class.  Refer to the API documentation in the base class for
        intended functionality.</p>
```

```
151
152         <p>This class uses the supplied agent pointer to setup the
153         list of agents managed and scheduled by this class.</p>
154
155         \param[in,out] agent A pointer to the agent to be registered.
156         This value is not used.
157
158         \return This method returns the iterator to the position of
159         the agent in its internal vector as a cross-reference to be
160         stored in an agent.
161     */
162     virtual void* addAgent(xxxx::Agent* agent);
163
164     /** Remove/unregister an agent with the event queue.
165
166         <p>This method implements the corresponding API method in the
167         class.  Refer to the API documentation in the base class for
168         intended functionality.</p>
169
170         <p>This method removes all events scheduled for the specified
171         agent in its internal data structures.</p>
172
173         \param[in,out] agent A pointer to the agent whose events are
174         to be removed from the vector managed by this class.
175     */
176     void removeAgent(xxxx::Agent* agent) override;
177
178     /** Determine if the event queue is empty.
179
180         This method implements the base class API to report if any
181         events are pending to be processed in the event queue.
182
183         \return This method returns true if the event queue of the
184         top-agent is logically empty.
185     */
186     virtual bool empty() {
187         return (agentList.empty() || top()->tier2->empty());
188     }
189
190     /** Obtain pointer to the highest priority (lowest receive time)
191         event.
192
193         This method can be used to obtain a pointer to the highest
194         priority event in this event queue, without de-queuing the
195         event.
196
197         \note The event returned by this method is not dequeued.
198
199         \return A pointer to the next event to be processed.  If the
200         queue is empty then this method returns NULL.
201     */
202     virtual xxxx::Event* front();
203
204     /** Method to obtain the next batch of events to be processed by
205         one agent.
206
207         <p>In XXXX agents are scheduled to process all events at a
208         given simulation time.  The next concurrent events (i.e.,
209         events with the same receive time) with the lowest time stamp
210         are to be placed in the supplied event container.  The event
211         container is then passed to the corresponding agent for
212         further processing.</p>
213
214         <p>This method essentially delegates the dequeue process to
215         the agent with the next lowest timestamp.  Once the agent has
216         been dequeued, this method fixes the heap by placing the
217         top-agent in its appropriate location in the heap.</p>
218
219         \param[out] events The event container in which the next set
220         of concurrent events are to be placed.  Note that the order of
221         concurrent events in the event container is unspecified.
222     */
223     virtual void dequeueNextAgentEvents(xxxx::EventContainer& events);
224
225     /** Enqueue a new event.
```



```
226
227         This method must be used to enqueue/add an event to this event
228         queue.  Once added the reference count on the event is
229         increased.  This method adds the event to the specified agent.
230         Next this method fixes the heap to ensure that the agent with
231         the least-time-stamp is at the top of the heap.  This method
232         essentially uses an internal helper method to accomplish its
233         tasks.
234
235         \param[in] agent The agent to which the event is to be
236         scheduled.  This agent corresponds to the agent ID returned by
237         event->getReceiverAgentID() method.
238
239         \param[in] event The event to be enqueued.  This parameter can
240         never be NULL.
241     */
242     virtual void enqueue(xxxx::Agent* agent, xxxx::Event* event);
243
244     /** Enqueue a batch of events.
245
246         This method can be used to enqueue/add a batch of events to
247         this event queue.  Once added the reference count on each one
248         of the events is increased.  This method provides a convenient
249         approach to enqueue a batch of events, particularly after a
250         rollback.  Next this method fixes the heap to ensure that the
251         agent with the least-time-stamp is at the top of the heap.
252         This method uses an internal helper method to accomplish its
253         tasks.
254
255         \param[in] agent The agent to which the event is to be
256         scheduled.  This agent corresponds to the agent ID returned by
257         event->getReceiverAgentID() method.  Currently, this value is
258         not used.
259
260         \param[in] event The list of events to be enqueued.  This
261         container can and will be be empty in certain situations.  The
262         reference counts of the events in the container remains
263         unmodified.  The list of events become part of the event
264         queue.
265     */
266     virtual void enqueue(xxxx::Agent* agent, xxxx::EventContainer& events);
267
268     /** Dequeue all events sent by an agent after a given time.
269
270         This method implements the base class API method.  This method
271         can be used to remove/erase events sent by a given agent after
272         a given simulation time.  This API is needed to cancel events
273         during a rollback.  Next this method fixes the heap to ensure
274         that the agent with the least-time-stamp is at the top of the
275         heap.
276
277         \param[in] dest The agent whose currently scheduled events
278         are to be checked and cleaned-up.  This agent must be a valid
279         agent that has been registered/added to this event queue.  The
280         pointer cannot be NULL.  This parameter is not used.
281
282         \param[in] sender The ID of the agent whose events have to be
283         removed.  This agent must be a valid agent that has been
284         registered/added to this event queue.  The pointer cannot be
285         NULL.
286
287         \param[in] sentTime The time from which the events are to be
288         removed.  All events (including those sent at this time) sent
289         by the sender agent are removed from this event queue.
290
291         \return This method returns the number of events actually
292         removed.
293     */
294     virtual int eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
295                            const xxxx::Time sentTime);
296
297     /** Print full contents of scheduler queue to given output stream.
298
299         This is a convenience method that is used primarily for
300         troubleshooting purposes.  This method prints all the events
```

```
301         in this queue, with each event on its own line.
302
303         \param[out] os The output stream to which the contents of the
304         queue are to be written.
305     */
306     virtual void prettyPrint(std::ostream& os) const;
307
308     /** Method to report aggregate statistics.
309
310         This method is invoked at the end of simulation after all
311         agents on this rank have been finalized.  This method can
312         report any aggregate statistics from the event
313         queue. Currently, this method does not have any additional
314         statistics to report.
315
316         \param[out] os The output stream to which the statistics are
317         to be written.
318     */
319     virtual void reportStats(std::ostream& os);
320
321 protected:
322     /** Enqueue a new event.
323
324         This method must be used to enqueue/add an event to this event
325         queue.  Once added the reference count on the event is
326         increased.  This method adds the event to the specified agent.
327         Next this method fixes the heap to ensure that the agent with
328         the least-time-stamp is at the top of the heap.
329
330         \param[in] agent The agent to which the event is to be
331         scheduled.  This agent corresponds to the agent ID returned by
332         event->getReceiverAgentID() method.
333
334         \param[in] event The event to be enqueued.  This parameter can
335         never be NULL.
336     */
337     virtual void enqueueEvent(xxxx::Agent* agent, xxxx::Event* event);
338
339     /** Convenience method to remove events.
340
341         This is an internal convenience method that is used to remove
342         the front (i.e., events with lowest timestamp) event list from this
343         queue.
344     */
345     void pop_front(xxxx::Agent* agent);
346
347     /** Convenience method to obtain the top-most or front agent.
348
349         This method can be used to obtain a pointer to the top/front
350         agent -- that is, the agent with the lowest timestamp event to
351         be scheduled next.
352
353         \return A pointer to the top-most agent in this heap.
354     */
355     inline xxxx::Agent* top() {
356         return agentList.front();
357     }
358
359     /** Convenience method to get the top-event time for a given
360         agent.
361
362         This method returns the top event time in the vector of events queue.
363         If agent's vector of event queue is empty, then it returns infinity.
364
365         \return The receive time of top event's recv time or
366         TIME_INFINITY if vector is empty.
367     */
368     inline xxxx::Time getTopTime(const xxxx::Agent* const agent) const {
369         return agent->tier2->empty() ? TIME_INFINITY :
370             agent->tier2->front()->getReceiveTime();
371     }
372
373     /** Comparator method to sort events in the heap.
374
375         This is the comparator method that is passed to various
```



```
376         standard C++ algorithms to organize events as a heap.  This
377         comparator method gives first preference to receive time of
378         events.  Tie between two events with the same recieve time is
379         broken based on the receiver agent ID.
380
381         \param[in] lhs The left-hand-side event to be used for
382         comparison.  This parameter cannot be NULL.
383
384         \param[in] rhs The right-hand-side event to be used for
385         comparison. This parameter cannot be NULL.
386
387         \return This method returns if lhs < rhs, i.e., the lhs event
388         should be scheduled before the rhs event.
389     */
390     inline bool compare(const Agent *lhs, const Agent * rhs) const {
391         return getTopTime(lhs) >= getTopTime(rhs);
392     }
393
394     /** Comparator method to sort events in the heap.
395
396         This is the comparator method that is passed to various
397         standard C++ algorithms to organize events as a heap.  This
398         comparator method gives first preference to receive time of
399         events.  Tie between two events with the same recieve time is
400         broken based on the receiver agent ID.
401
402         \param[in] lhs The left-hand-side event to be used for
403         comparison.  This parameter cannot be NULL.
404
405         \param[in] rhs The right-hand-side event to be used for
406         comparison. This parameter cannot be NULL.
407
408         \return This method returns if lhs < rhs, i.e., the lhs event
409         should be scheduled before the rhs event.
410     */
411     inline static bool lessThan(const HOETier2Entry& lhs,
412                                 const xxxx::Event* const event) {
413         return (lhs.getReceiveTime() < event->getReceiveTime());
414     }
415
416     /** Comparator method to sort events in the heap.
417
418         This is the comparator method that is passed to various
419         standard C++ algorithms to organize events as a heap.  This
420         comparator method gives first preference to receive time of
421         events.  Tie between two events with the same recieve time is
422         broken based on the receiver agent ID.
423
424         \param[in] lhs The left-hand-side event to be used for
425         comparison.  This parameter cannot be NULL.
426
427         \param[in] rhs The right-hand-side event to be used for
428         comparison. This parameter cannot be NULL.
429
430         \return This method returns if lhs < rhs, i.e., the lhs event
431         should be scheduled before the rhs event.
432     */
433     inline static bool lessThanPtr(const HOETier2Entry* const lhs,
434                                    const xxxx::Event* const event) {
435         return (lhs->getReceiveTime() < event->getReceiveTime());
436     }
437
438     /** The getNextEvents method.
439
440         This method is a helper that will grab the next set of events
441         to be processed for a given agent.  This method is invoked in
442         dequeueNextAgentEvents() method in this class.
443
444         \param[out] container The reference of the container into
445         which events should be added.
446     */
447     void getNextEvents(Agent* agent, EventContainer& container);
448
449         /** Obtain the current index of the agent from it's
450         cross-reference.
```

```
451
452         This method is a refactored utility method that has been
453         introduced to streamline the code.  This method essentially
454         obtains the index position of the given agent in the agentList
455         vector from the agent's fibHeapPtr corss-reference.  This
456         cross-reference is consistently updated by the various methods
457         in this class to enable rapid access to the location of the
458         agent.
459
460         \param[in] agent The agent whose index value in the agentList
461         is to be determined.
462
463         \return The index position of the agent in the agentList
464         vector (if all checks pass).
465     */
466     size_t getIndex(xxxx::Agent *agent) const;
467
468     /** Update position of agent in the scheduler's heap.
469
470         This is an internal helper method that is used to update the
471         position of an agent in the scheduler's heap.  This method
472         essentially performs sanity checks, uses the fixHeap() method
473         to update position of the agent, and updates cross references
474         for future use.
475
476         \param[in] agent The agent whose position in the heap is to be
477         updated.  This pointer cannot be NULL.
478
479         \return This method returns the updated index position of the
480         agent in agentList (the vector that serves as storage for the
481         heap).
482     */
483     size_t updateHeap(xxxx::Agent* agent);
484
485     /** Fix-up the location of the agent in the heap.
486
487         This method can be used to update the location of an agent in
488         the heap.
489
490         \note The implementation for this method has been heavily
491         borrowed from libstdc++'s code base to ensure that heap
492         updates are consistent with std::make_heap API.
493         Unfortunately, this does imply that there is a chance this
494         method may be incompatible with future versions.
495
496         \param[in] currPos The current position of the agent in the
497         heap whose position is to be updated.  This value is the index
498         position of the agent in the agentList vector.
499
500         \return This method returns the new position of the agent in
501         the agentList vector.
502     */
503     size_t fixHeap(size_t currPos);
504
505     /** Convenience method to determine if an event is a future event.
506
507         This method is a helper method used in the eraseAfter() method
508         to determine if a given event is a future event from a given
509         sender agent.
510
511         \param[in] sender The sender agent to be used in comparison.
512
513         \param[in] sendTime The reference time for comparison
514
515         \param[in] evt The event to be checked if it is future event.
516
517         \return This method returns true if the event is sent from a
518         given sender agent and its send time is greater-or-equal to
519         the given sentTime.
520      */
521     inline bool
522     isFutureEvent(const xxxx::AgentID sender, const xxxx::Time sentTime,
523                   const xxxx::Event* evt) const {
524         return ((evt->getSenderAgentID() == sender) &&
525                 (evt->getSentTime() >= sentTime));
```



```
526     }
527
528     /** Helper method to reuse tier2 entries or create a new one.
529
530         This method is a convenience method to recycle tier2 entry
531         object is available.  If the recycle bin is empty, then this
532         method creates a new object.
533
534         \param[in] event The event to be used to initialize and to be
535         contained in the newly created tier2 entry.
536
537         \return A tier2 entry initialized and containing the given
538         event.
539      */
540     HOETier2Entry* makeTier2Entry(xxxx::Event* event) {
541         if (!tier2Recycler.empty()) {
542             HOETier2Entry* entry = tier2Recycler.back();
543             tier2Recycler.pop_back();
544             entry->reset(event);
545             return entry;
546         }
547         return new HOETier2Entry(event);
548     }
549
550 private:
551     /** The backing storage for events managed by this class.
552
553         This vector contains the list of agents being managed by the
554         class.  The agents in the vector are stored and maintained as
555         a heap.  The heap is created and managed using standard C++
556         algorithms, namely: \c std::make_heap, \c std::push_heap, and
557         \c std::pop_heap.
558     */
559     std::vector<xxxx::Agent*> agentList;
560
561     /** Stats object to track the average tier-2 bucket size.  This
562         value is the one that primarily determines if tier-2
563         operations are going to be optimal or not.  Higher this value,
564         the better for this event queue.
565     */
566     Avg avgSchedBktSize;
567
568     /** The number of times the fixHeap method performed heap-fixing
569         operations.  This variable is fine-grained in that it
570         accumulates the average number of compares that occur to fix
571         up the heap of agents.  The fixHeap method has a q1 = O(log
572         n1) compares.  So if this method is called m times, the
573         statistics reports (q1 + q2 + ... + qm) / m.
574     */
575     Avg fixHeapSwapCount;
576
577     /** The average queue size for each agent.  This value determines
578         the time takes to find a bucket into which an event is to be
579         inserted.
580     */
581     Avg agentBktCount;
582
583     /** A stack to recycle Tier2 entries to minimize memory allocation
584         calls for these small blocks used in this queue.
585     */
586     std::deque<HOETier2Entry*> tier2Recycler;
587 };
588
589 END_NAMESPACE(xxxx)
590
591 #endif
```

```cpp
#ifndef THREE_TIER_HEAP_EVENT_QUEUE_CPP
#define THREE_TIER_HEAP_EVENT_QUEUE_CPP

#include "ThreeTierHeapEventQueue.h"
#include <algorithm>

BEGIN_NAMESPACE(xxxx)

// A convenience shortcut used just in this source file
using Tier2List = std::deque<HOETier2Entry*>;
// using Tier2List = std::vector<Tier2Entry>;

ThreeTierHeapEventQueue::ThreeTierHeapEventQueue() :
    EventQueue("HeapOfVectorsEventQueue") {
    // Nothing else to be done.
}

ThreeTierHeapEventQueue::~ThreeTierHeapEventQueue() {
    // Clear up memory allocated for HOETier2Entry
    for (HOETier2Entry* entry : tier2Recycler) {
        delete entry;
    }
}

void*
ThreeTierHeapEventQueue::addAgent(xxxx::Agent* agent) {
    agentList.push_back(agent);
    // Create the vector that is used to manage events for the agent.
    agent->tier2 = new Tier2List();
    return reinterpret_cast<void*>(agentList.size() - 1);
}

void
ThreeTierHeapEventQueue::removeAgent(xxxx::Agent* agent) {
    ASSERT( agent != NULL );
    ASSERT(!empty());
    // Decrease reference count for all events in the agent event queue
    // before agent removal.
    ASSERT( agent->tier2 != NULL );
    // Logically remove events in this agent's tier2 queues/buckets
    Tier2List& tier2eventPQ = *agent->tier2;
    for (xxxx::HOETier2Entry* bucket : tier2eventPQ) {
        for (Event* evt : bucket->getEventList()) {
            evt->decreaseReference();  // logically remove event
        }
        // Free the memory reserved for this bucket
        delete bucket;
    }
    // Clear out tier2 queue (so this agent's time becomes PINFINITY)
    agent->tier2->clear();
    // Update the heap to place agent with LTSF
    updateHeap(agent);
}

xxxx::Event*
ThreeTierHeapEventQueue::front() {
    return (!top()->tier2->empty()) ? top()->tier2->front()->getEvent() : NULL;
}

void
ThreeTierHeapEventQueue::pop_front(xxxx::Agent* agent) {
    // Decrease reference count for all events in the front of the
    // agent event queue before the list of events is removed from the
    // event queue.
    std::vector<xxxx::Event*>& eventList =
        agent->tier2->front()->getEventList();
    for (Event* evt: eventList) {
        evt->decreaseReference();
    }
    // agent->tier2->erase(agent->tier2->begin());
    tier2Recycler.emplace_back(agent->tier2->front());
    agent->tier2->pop_front();
}

void
```



```cpp
ThreeTierHeapEventQueue::getNextEvents(Agent* agent,
                                       EventContainer& container) {
    ASSERT(container.empty());
    ASSERT(agent->tier2 != NULL);
    Tier2List& tier2 = *agent->tier2;
    ASSERT(tier2.front()->getEvent() != NULL);
    // All events in tier2 front should have same receive times
    const xxxx::Time eventTime = tier2.front()->getReceiveTime();
    // Copy all the events out of the tier2 front into the return contianer
    // container = std::move(agent->tier2->front().getEventList());
    std::vector<xxxx::Event*>& evtList = tier2.front()->getEventList();
    container.assign(evtList.begin(), evtList.end());
    DEBUG({
            // Do validation checks on the events in tier2
            for (const Event* event : container) {
                //  All events must have the same receive time
                ASSERT( event->getReceiveTime() == eventTime );

                // We should never process an anti-message.
                if (event->isAntiMessage()) {
                    std::cerr << "Anti-message Processing: " << *event
                              << std::endl;
                    std::cerr << "Trying to process an anti-message event, "
                              << "please notify XXXX developers of this issue"
                              << std::endl;
                    abort();
                }
                // Ensure that the top event is greater than LVT
                if (event->getReceiveTime() <= agent->getTime(Agent::LVT)) {
                    std::cerr << "Agent is being scheduled to process "
                              << "an event ("
                              << *event << ") that is at or below it LVT (LVT="
                              << agent->getTime(Agent::LVT) << ", GVT="
                              << agent->getTime(Agent::GVT)
                              << "). This is a serious error. Aborting.\n";
                    std::cerr << *agent << std::endl;
                    abort();
                }
                // Ensure reference counts are consistent.
                ASSERT(event->getReferenceCount() < 3);
                DEBUG(std::cout << "Delivering: " << *event << std::endl);
            }
        });
    // Recycle the entry at the beginning of the queue.
    tier2Recycler.emplace_back(tier2.front());
    tier2.pop_front();
    // std::rotate(tier2.begin(), tier2.begin() + 1, tier2.end());
    // tier2.pop_back();
    // Track bucket/block size statistics
    avgSchedBktSize += container.size();
}

void
ThreeTierHeapEventQueue::dequeueNextAgentEvents(xxxx::EventContainer& events) {
    if (!empty()) {
        // Get agent and validate.
        xxxx::Agent* const agent = top();
        ASSERT(agent != NULL);
        ASSERT(getIndex(agent) == 0);
        // Have the events give up its next set of events
        getNextEvents(agent, events);
        ASSERT(!events.empty());
        // Fix the position of this agent in the scheduler's heap.
        updateHeap(agent);
    }
}

void
ThreeTierHeapEventQueue::enqueue(xxxx::Agent* agent, xxxx::Event* event) {
    // Use helper method (just below this one) to add event and fix-up
    // the queue.  First Increase event reference count for every
    // event added to the event queue.
    ASSERT( event->getReferenceCount() < 2 );
    event->increaseReference();
    enqueueEvent(agent, event);
```

```
151     updateHeap(agent);
152 }
153
154 void
155 ThreeTierHeapEventQueue::enqueueEvent(xxxx::Agent* agent, xxxx::Event* event) {
156     ASSERT(agent != NULL);
157     ASSERT(event != NULL);
158     ASSERT( agent->tier2 != NULL );
159     ASSERT(getIndex(agent) < agentList.size());
160     // A convenience reference to tier2 list of buckets
161     Tier2List& tier2 = *agent->tier2;
162     // Use binary search O(log n) to find match or insert position
163     agentBktCount += tier2.size();
164     Tier2List::iterator iter =
165         std::lower_bound(tier2.begin(), tier2.end(), event, lessThanPtr);
166     // There are 3 cases: 1. we found matching bucket, 2: iterator
167     // to bucket with higher recvTime, or 3: tier2.end().
168     if (iter == tier2.end()) {
169         tier2.emplace_back(makeTier2Entry(event));  // add new entry to end.
170     } else if ((*iter)->getReceiveTime() == event->getReceiveTime()) {
171         // We found an existing bucket. Append this event to this
172         // existing bucket.
173         (*iter)->updateContainer(event);
174     } else {
175         // If there is no bucket with a matching receive time in Tier2
176         // vector, then insert an instance of HOETier2Entry (aka
177         // bucket) into the vector at the appropriate position.
178         ASSERT((*iter)->getReceiveTime() > event->getReceiveTime());
179         tier2.emplace(iter, makeTier2Entry(event));
180     }
181     // ASSERT(std::is_sorted(tier2.begin(), tier2.end()));
182 }
183
184 void
185 ThreeTierHeapEventQueue::enqueue(xxxx::Agent* agent,
186                                  xxxx::EventContainer& events) {
187     ASSERT(agent != NULL);
188     // Note: events container may be empty!
189     ASSERT(getIndex(agent) < agentList.size());
190     // Add all events to tier2 entries appropriately.
191     for (xxxx::Event* event : events) {
192         // Enqueue event but don't waste time fixing-up heap yet for
193         // this agent.  We will do it at the end after all events are
194         // added.  However, we don't increase reference counts in this
195         // API.
196         enqueueEvent(agent, event);
197     }
198     // Clear out all the events in the incoming container
199     events.clear();
200     // Update the location of this agent on the heap as needed.
201     updateHeap(agent);
202 }
203
204 int
205 ThreeTierHeapEventQueue::eraseAfter(xxxx::Agent* dest,
206                                     const xxxx::AgentID sender,
207                                     const xxxx::Time sentTime) {
208     int  numRemoved = 0;
209     ASSERT( dest->tier2 != NULL );
210     Tier2List& tier2eventPQ = *dest->tier2;
211     long currIdx = tier2eventPQ.size() - 1;
212     while (!tier2eventPQ.empty() && (currIdx >= 0)) {
213         if (tier2eventPQ[currIdx]->getReceiveTime() > sentTime) {
214             std::vector<xxxx::Event*>& eventList =
215                 tier2eventPQ[currIdx]->getEventList();
216             size_t index = 0;
217             while (!eventList.empty() && (index < eventList.size())) {
218                 Event* const evt = eventList[index];
219                 ASSERT(evt != NULL);
220                 if (isFutureEvent(sender, sentTime, evt)) {
221                     evt->decreaseReference();
222                     numRemoved++;
223                     eventList[index] = eventList.back();
224                     eventList.pop_back();
225                 } else {
```



```
226                     index++;  // onto next event in this bucket
227                 }
228             }
229             // If all events are canceled then this bucket needs to be
230             // removed from the tier2 entry.
231             if (eventList.empty()) {
232                 tier2Recycler.emplace_back(tier2eventPQ[currIdx]);
233                 tier2eventPQ.erase(tier2eventPQ.begin() + currIdx);
234             }
235         }
236         currIdx--;
237     }
238     // Update the 1st tier heap for scheduling.
239     updateHeap(dest);
240     // Return number of events canceled to track statistics.
241     return numRemoved;
242 }
243
244 void
245 ThreeTierHeapEventQueue::reportStats(std::ostream& os) {
246     UNUSED_PARAM(os);
247     const long comps = std::log2(agentList.size()) *
248         avgSchedBktSize.getCount() + fixHeapSwapCount.getSum();
249     os << "Average #buckets per agent  : " << agentBktCount    << std::endl;
250     os << "Average scheduled bucket size: " << avgSchedBktSize  << std::endl;
251     os << "Average fixHeap compares    : " << fixHeapSwapCount << std::endl;
252     os << "Compare estimate            : " << comps            << std::endl;
253 }
254
255 void
256 ThreeTierHeapEventQueue::prettyPrint(std::ostream& os) const {
257     os << "HeapOfVectorsEventQueue::prettyPrint() : not implemented.\n";
258 }
259
260 size_t
261 ThreeTierHeapEventQueue::getIndex(xxxx::Agent *agent) const {
262     ASSERT(agent != NULL);
263     size_t index = reinterpret_cast<size_t>(agent->fibHeapPtr);
264     ASSERT(index < agentList.size());
265     ASSERT(agentList[index] == agent);
266     return index;
267 }
268
269 size_t
270 ThreeTierHeapEventQueue::updateHeap(xxxx::Agent* agent) {
271     ASSERT(agent != NULL);
272     size_t index = getIndex(agent);
273     if (agent->oldTopTime != getTopTime(agent)) {
274         index = fixHeap(index);
275         // Update the position of the agent in the scheduler's heap
276         // Validate
277         ASSERT(agentList[index] == agent);
278         ASSERT(getIndex(agent) == index);
279         // Update time value as well for future access
280         agent->oldTopTime = getTopTime(agent);
281         // Validation check.
282         ASSERT(getTopTime(agentList[0]) <= getTopTime(agentList[1]));
283     }
284     // Return the new index position of the agent
285     return index;
286 }
287
288 size_t
289 ThreeTierHeapEventQueue::fixHeap(size_t currPos) {
290     ASSERT(currPos < agentList.size());
291     xxxx::Agent* value    = agentList[currPos];
292     const size_t len      = (agentList.size() - 1) / 2;
293     size_t secondChild    = currPos;
294     int          opCount  = 0;
295     // This code was borrowed from libstdc++ implementation to ensure
296     // that the fix-ups are consistent with std::make_heap API.
297     while (secondChild < len)  {
298         secondChild = 2 * (secondChild + 1);
299         if (compare(agentList[secondChild], agentList[secondChild - 1])) {
300             secondChild--;
```

```
301         }
302         agentList[currPos] = std::move(agentList[secondChild]);
303         agentList[currPos]->fibHeapPtr = reinterpret_cast<void*>(currPos);
304         currPos = secondChild;
305         opCount++;  // track statistics on number of operations performed
306     }
307     if (((agentList.size() & 1) == 0) &&
308         (secondChild == (agentList.size() - 2) / 2)) {
309         secondChild        = 2 * (secondChild + 1);
310         agentList[currPos] = std::move(agentList[secondChild - 1]);
311         agentList[currPos]->fibHeapPtr = reinterpret_cast<void*>(currPos);
312         currPos            = secondChild - 1;
313         opCount++;  // track statistics on number of operations performed
314     }
315     // Use libstdc++'s internal method to fix-up the vector from the
316     // given location.
317     // std::__push_heap(agentList.begin(), currPos, 0, value,
318     //                  __gnu_cxx::__ops::__iter_comp_val(compare));
319
320     size_t parent = (currPos - 1) / 2;
321     while ((currPos > 0) && (compare(agentList[parent],value))) {
322         agentList[currPos] = std::move(agentList[parent]);
323         agentList[currPos]->fibHeapPtr = reinterpret_cast<void*>(currPos);
324         currPos = parent;
325         parent  = (currPos - 1) / 2;
326         opCount++;  // track statistics on number of operations performed
327     }
328     agentList[currPos] = value;
329     agentList[currPos]->fibHeapPtr = reinterpret_cast<void*>(currPos);
330     // Update aggregate statistics
331     fixHeapSwapCount += opCount;
332     // Return the final index position for the agent
333     return currPos;
334 }
335
336 END_NAMESPACE(xxxx)
337
338 #endif
```

```
#ifndef TWO_TIER_LADDER_QUEUE_H
#define TWO_TIER_LADDER_QUEUE_H

#include <forward_list>
#include <queue>
#include <vector>
#include <typeinfo>
#include <set>
#include "Avg.h"
#include "Event.h"
#include "EventQueue.h"

/** \file LadderQueue.h

    \brief Enhancement of LadderQueue to improve performance of
    Optimistic Parallel Simulations by minimizing rollbacks.

    The LadderQueue data structure is detailed in the following paper:

    W. Tang, R. Goh, and I. Thng, "Ladder queue: An O(1) priority
    queue structure for large-scale discrete event simulation", ACM
    TOMACS, Vol 15, Issue 3, Pages 175--204, July 2005. URL:
    http://doi.acm.org/10.1145/1103323.1103324

    <p>One major disadvantage of the LadderQueue is that canceling
    events due to a rollback is expensive -- the whole queue has to be
    scanned.</p>

    <p>In order to reduce the overhead of scanning events for
    canceling, this TwoTierLadderQueue further subdivides each bucket
    in Top and ladder Rung to store events in a 2nd Tier based on
    their sender's ID.  It uses a simple hash function on the sender's
    AgentID to identify the 2nd tier bucket and enqueue's the event
    into that bucket.  Currently, the hash function is simply
    implemented as a modulo t2k, with t2k being an implementation
    dependent value.  Since second tier buckets (implemented as
    std::vector) are preallocated, Small t2k values increase 2nd tier
    bucket sizes increasing time for cancellation.  On the other hand
    if buckets are not used, then the space/time invested to create
    them can become an overhead.</p>

    <p>Note that the TwoTierLadderQueue would have very similar
    characteristics to LadderQueue in sequential or 1 process
    simulation as there are no rollbacks</p>
*/

// Bucket size after which new rung is created in ladder
#define LQ2T_THRESH 50

/** \def LQ2T_STATS(x)

    \brief Define a convenient macro for conditionally compiling
    additional statistics collection regarding ladder queue.

    Define a custom macro LQ2T_STATS (note the all caps) macro to be
    used to conditionally compile in debugging code to generate
    detailed logs.  This helps to minimize code modification to insert
    and remove debugging messages.
*/
#define COMMA ,
#define LQ2T_STATS(x) x
// #define LQ2T_STATS(x)

BEGIN_NAMESPACE(xxxx)

/** A convenience alias for list of events maintained by a sub-bucket. */
using BktEventList = std::vector<xxxx::Event*>;

/** Alias to the data structure for holding a vector of sub-buckets in
    a TwoTierBucket.
*/
using SubBucketList = std::vector<BktEventList>;

constexpr bool SenderID   = true;
constexpr bool ReceiverID = false;
```



```

/** A generic two tier bucket that is used for both Top and Rungs of
    the 2-tier ladderQ.

    <p>This bucket does not store events in it directly.  Instead it
    splits into t2k sub-buckets based on a simple hash function.
    Currently, the hash function is simply implemented as a modulo
    t2k, with t2k being an implementation dependent value.  Splitting
    the events into sub-buckets makes event cancellations easier.
    </p>
*/
class TwoTierBucket {
public:
    /** The shared parameter indicating the number of sub-buckets to
        be used in each 2-tier bucket.  This value defaults to 32.  It
        is overriden by by command-line argument when ladder queue is
        used.
    */
    static int t2k;

    /** Constructor to create a bucket with fixed number (i.e., t2k)
        of tier-2 lists.
    */
    TwoTierBucket() : subBuckets(t2k), count(0) {}

    /** A move constructor to facilitate moving objects (if needed).

        \param[in,out] src The source object whose data is to be moved
        into this.  The source object does not contain any useful
        information after the move is complete.
    */
    TwoTierBucket(TwoTierBucket&& src) : subBuckets(std::move(src.subBuckets)),
                                         count(std::move(src.count)) {
        // Reset count in source to aid debugging.
        src.count = 0;
    }

    /** The destructor for this class.

        The destructor decreases the reference count on all the events
        in its list to free-up any pending events.
    */
    ~TwoTierBucket();

    /** The hash function used to distribute events into sub-buckets.

        \param[in] sender The sender's ID to be hashed.

        \return The hash value based on sender ID.  The return value,
        say hash, must be in the range 0 <= hash < t2k.
    */
    inline int hash(const xxxx::AgentID sender) const {
        // Use a simple hashing function for now.
        return (sender % t2k);
    }

    /** Add an event to this TwoTier bucket based on sender's ID

        This method is a template specialization to use the sender's
        ID for hashing to find sub-bucket. The event is added to the
        sub-bucket identified using the hash function in this class.

        \param[in] event The event to be added to this bucket.  This
        method does not alter the refernece counts on events (as the
        top-level TwoTierLadderQueue performs the reference count
        management).
    */
    template <bool Sendr, typename std::enable_if<Sendr>::type* = nullptr>
    void push_back(xxxx::Event* event) {
        const size_t subBktIdx = hash(event->getSenderAgentID());
        subBuckets[subBktIdx].push_back(event);
        count++;
    }

    /** Add an event to this TwoTier bucket based on receiver's ID
```

```
151
152         This method is a template specialization to use the receiver's
153         ID for hashing to find sub-bucket. The event is added to the
154         sub-bucket identified using the hash function in this class.
155
156         \param[in] event The event to be added to this bucket.  This
157         method does not alter the refernece counts on events (as the
158         top-level TwoTierLadderQueue performs the reference count
159         management).
160     */
161     template <bool Recvr, typename std::enable_if<!Recvr>::type* = nullptr>
162     void push_back(xxxx::Event* event) {
163         const size_t subBktIdx = hash(event->getReceiverAgentID());
164         subBuckets[subBktIdx].push_back(event);
165         count++;
166     }
167
168     /** Move all the events from the given two tier bucket into this
169         bucket.
170
171         This method moves all the events from the sub-buckets in
172         srcBkt to corresponding sub-buckets in this.
173
174         \param[in,out] srcBkt The bucket from where the events are to
175         be moved into this bucket.
176     */
177     void push_back(TwoTierBucket&& srcBkt);
178
179     /** Helper method to move all events from a 2-tier bucket into a
180         single list of events.
181
182         This method is a convenience method that is used by the bottom
183         tier to combine all the events from various sub-buckets into a
184         single list of events.
185
186         \param[out] dest The destination event list to which all the
187         events are to added.
188
189         \param[in,out] srcBkt The bucket from where the events are to
190         be moved.  After this call the srcBkt will not have any events
191         in it.
192     */
193     static void push_back(BktEventList& dest, TwoTierBucket&& srcBkt);
194
195     /** Obtain the count of events which includes the events in sub-buckets.
196
197         \return The sum of events in all of the sub-buckets.
198     */
199     size_t size() const {
200         ASSERT( getEventCount() == count );
201         return count;
202     }
203
204     /** Convenience method to determine if the bucket is empty.
205
206         \return This method returns true if this bucket does not
207         contain any events in it.
208      */
209     bool empty() const {
210         ASSERT( getEventCount() == count );
211         return (count == 0);
212     }
213
214     /** Convenience method to remove all events sent by the sender
215         at-or-after the given send Time.
216
217         This method removes computes the sub-bucket that contains all
218         the events for this sender and removes events from that
219         sub-bucket.
220
221         \param[in] sender The sender agent whose events are to be
222         removed.
223
224         \param[in] sendTime The time at-or-after which events from the
225         sender are to be removed from the given list.
```



```
226
227         \param[in,out] scans Statistics object (if stats is enabled)
228         to track number of events scanned.
229
230         \return This method returns the number of events that were
231         removed.
232     */
233     int remove_after(xxxx::AgentID sender, const Time sendTime
234                      LQ2T_STATS(COMMA Avg& scans));
235
236     /** Remove all events in this bucket for a given receiver agent
237         ID.
238
239         This is a convenience method that removes all events for a
240         given receiver agent in this bucket.  This method is used to
241         remove events scheduled for an agent, when an agent is removed
242         from the scheduler.  This method has to search through all the
243         sub-buckets because the condition is based on receiver (and
244         not sender).
245
246         \param[in] receiver The receier ID whose events are to be
247         removed from the sub-buckets.
248
249         \return This method returns the number of events removed.
250     */
251     int remove(xxxx::AgentID receiver);
252
253     /** This method is purely for troubleshooting one scenario
254         where an event would get stuck in the ladder and not get
255         scheduled correctly.
256
257         \param[in] recvTime The time to be used for checking to see if
258         sub-buckets have an event before this time.
259
260         \return Returns true if an event before this receiveTime (for
261         any agent) is pending in a sub-bucket.
262     */
263     bool haveBefore(const Time recvTime) const;
264
265     /** Convenience method to remove all events from a given sender
266         that were sent at-or-after the given sendTime.
267
268         This method linearly scans the given event list, checks, and
269         removes all events that were sent by the sender at-or-after
270         the specified send time.
271
272         \note This method assumes unsorted list of events and does not
273         preserve order of events if an event is cancelled -- this is
274         because Events to be removed are moved to the back and popped
275         to reduce deletion time.
276
277         \param[in,out] list The list of events from where all events
278         for the sender are to be removed.  This method linearly scans
279         through this list.  If events are removed, the order of events
280         in the list is not preserved.
281
282         \param[in] sender The sender agent whose events are to be
283         removed.
284
285         \param[in] sendTime The time at-or-after which events from the
286         sender are to be removed from the given list.
287
288         \return This method returns the number of events that were
289         removed.
290     */
291     static int remove_after(BktEventList& list, xxxx::AgentID sender,
292                             const xxxx::Time sendTime);
293
294     /** Remove all events in a given event list for a given receiver
295         agent ID.
296
297         This is a convenience/helper method that removes all events
298         for a given receiver agent in a given sub-bucket/list.  This
299         method is used to remove events scheduled for an agent, when
300         an agent is removed from the scheduler.  This method has to
```

```
301         search through all the events in the list to remove them.
302
303         \param[in] subBkt The sub-bucket or list from where events are
304         to be removed.
305
306         \param[in] receiver The receier ID whose events are to be
307         removed from the sub-buckets.
308
309         \return This method returns the number of events removed.
310     */
311     static int remove(BktEventList& list, xxxx::AgentID receiver);
312
313     /** Obtain a reference to the list of sub-buckets in this bucket.
314
315         \return Mutable reference to the list of sub-buckets in this
316         bucket.
317     */
318     SubBucketList& getSubBuckets() { return subBuckets; }
319
320     /** Convenience method to reset count of events in this bucket to
321         zero.
322
323         This method is used in TwoTierRung operations to reset the
324         events in this bucket to zero, after events have been moved
325         out of this bucket.
326     */
327     void resetCount() {
328         count = 0;
329     }
330
331 protected:
332     /** Return sum of events in each sub-bucket.
333
334         This method is used purely for validation/debugging.  This
335         method iterates over each sub-bucket in the list and returns
336         the actual count of events.  This value must be consistent
337         with the value in the count instance variable.
338
339         \return The actual sum of events in various sub-buckets.
340     */
341     size_t getEventCount() const;
342
343     /** Clear out all the events in this bucket.
344
345         This method clears out all the events in various sub-buckets
346         in this 2-tier bucket.  It also sets count to zero.
347
348         \note This method decreases references on any pending events.
349     */
350     void clear();
351
352 private:
353     /** The list of tier-2 sub-buckets that contain events distributed
354         based on the hash of the receiver's ID.
355     */
356     SubBucketList subBuckets;
357
358     /** The total number of events in all of the sub-buckets.  This
359         information is primary used to quickly respond to the size()
360         method calls
361     */
362     size_t count;
363 };
364
365 /** The class that forms the Top rung of a 2-tier ladder queue.
366
367     The top-rung of the 2-tier ladder queue behaves similar to the
368     ladder queue with respect to managing time stamps.  However, the
369     organization is different -- events are not stored in a linear
370     list.  Instead they are stored in sub-buckets based on a hash of
371     the sender agent's ID.
372
373     \note Do not call push_back directly.  Instead use the add method
374     in this class to add events.
375 */
```



```
376 class TwoTierTop : public TwoTierBucket {
377     friend class TwoTierRung;
378     friend class TwoTierLadderQueue;
379 public:
380     /** Construct and initialize top to empty state.
381
382         The constructor uses a convenience method in this class to
383         reset the timestamps to zero.
384     */
385     TwoTierTop() {
386         reset();
387     }
388
389     /** The destructor
390
391         Currently the destructor has nothing much to do as the base
392         class does all of the necessary clean-ups.
393      */
394     ~TwoTierTop() {}
395
396     /** Method to add events to top and update current minimum and
397         maximum time stamp values.
398
399         \param[in] event The event to be added to the top.  This
400         pointer cannot be NULL.
401     */
402     void add(xxxx::Event* event);
403
404     /** Return the current start-time for top.
405
406         \note This value changes when events are added/removed. So
407         don't think about caching this value.
408
409         \return The current start time. This value is used for
410         scheduling events and creating rungs.
411     */
412     Time getStartTime() const { return topStart; }
413
414     /** Returns the minimum timestamp of events in this rung.
415
416         \note This value changes when events are added/removed. So
417         don't think about caching this value.
418
419         \return The minimum timestamp of events in this rung.
420     */
421     Time getMinTime() const { return minTS; }
422
423     /** Returns the maximum timestamp of events in this rung.
424
425         \note This value changes when events are added/removed. So
426         don't think about caching this value.
427
428         \return The maximum event timestamp in this rung.
429     */
430     Time getMaxTime() const { return maxTS; }
431
432     /** Convenience method to determine if given time is within the
433         <i>current</i> minmum and maximum time.
434
435         \param[in] ts The timestamp value to be checked.
436
437         \return This method returns true if getMinTime() <= ts <=
438         getMaxTime().  Otherwise it returns false.
439      */
440     bool contains(const Time ts) const {
441         return (ts >= minTS) && (ts <= maxTS);
442     }
443
444     /** Convenience method compute the bucket size for the top-level
445         rung of the TwoTierLadder queue.
446
447         \return The suggested bucket width (in terms of time) for the
448         top-level rung of the TwoTierLadder.
449     */
450     double getBucketWidth() const {
```

```
451         DEBUG(std::cout << "minTS=" << minTS << ", maxTS=" << maxTS
452               << ", size=" << size() << std::endl);
453         return std::max((maxTS - minTS + size() - 1.0) / size(), 0.01);
454     }
455
456 protected:
457     /** Helper method to reset top either during construction or
458         whenever it is emptied to move events into the ladder.
459
460         \param[in] topStart An optional start time for the top rung.
461     */
462     void reset(const Time topStart = 0);
463
464 private:
465     /** Instsance variable to track the current minimum timestamp of
466         events in top.  This value changes each time a new event is
467         added to the top via the add emthod.
468     */
469     xxxx::Time minTS;
470
471     /** Instsance variable to track the current maximum timestamp of
472         events in top.  This value changes each time a new event is
473         added to the top via the add emthod.
474     */
475     xxxx::Time maxTS;
476
477     /** Instsance variable to track the last time top was reset.  This
478         is used for debugging/troubleshooting purposes.
479     */
480     xxxx::Time topStart;
481 };
482
483 /** The bottom most rung of the TwoTierLadder queue.  The bottom rung
484     is the same as that of the standard ladder queue.  However, in
485     2-tier ladder queue, the size of the bottom has been relaxed.  So
486     bottom can be pretty long.  This implies that 2-tier ladder queue
487     will not be O(1).  It will be O(n log n).  However, it should
488     perform just fine as the ladder queue.
489
490     \note In XXXX we have an API requirement/guarantee that all the
491     concurrent events we have will be scheduled simultaneously.  This
492     eases agent development in many applications.  Consequently, it is
493     imperative that bottom be allowed to be long to contain all
494     concurrent events.
495
496     \note Do not use front() / back() to access the first event in
497     bottom. Instead use the first_event() method.
498 */
499 class OneTierBottom : public BktEventList {
500 public:
501     /** The default and only constructor.  It does not have any
502         special work to do as the base class handles most of the
503         tasks.
504     */
505     OneTierBottom() {}
506
507     /** Add events from a TwoTierBucket into the bottom.
508
509         This method is used to bulk move events from a rung of the
510         ladder (or top) into the bottom.  The events are added and
511         sorted in preparation for scheduling.
512
513         \param bucket The 2-tier bucket from where events are to be
514         moved into the bottom rung.
515     */
516     void enqueue(TwoTierBucket&& bucket);
517
518     /** Add a single event to the bottom rung.
519
520         This method uses binary-search (O(log n)) to insert an event
521         into the bottom.
522
523         \param[in] event The event to be added to the bottom rung.
524         This pointer cannot be NULL.  No operations are done on the
525         reference-counters in this method.
```



```
526     */
527     void enqueue(xxxx::Event* event);
528
529     /** Convenience method to dequeue events after a given time.
530
531         \param[in] sender The sender agent whose events are to be
532         removed.
533
534         \param[in] sendTime The time at-or-after which events from the
535         sender are to be removed from the given list.
536     */
537     int remove_after(xxxx::AgentID sender, const Time sendTime);
538
539     /** Remove all events for a given receiver agent in the bucket
540         encapsulated by this object.
541
542         This is a convenience method that removes all events for a
543         given receiver agent in this object.  This method is used to
544         remove events scheduled for an agent, when an agent is removed
545         from the scheduler.
546     */
547     int remove(xxxx::AgentID receiver) {
548         // Use static convenience method do to this task.
549         return TwoTierBucket::remove(*this, receiver);
550     }
551
552     /** Convenience method used to dequee the next set of events for
553         scheduling.
554
555         This method is used to provide necessary implemntation to
556         interface with the XXXX scheduler.  This method dequeues the
557         next batch of the concurrent events for processing by a given
558         agent.
559
560         \param[out] events The container to which all the events to be
561         processed is to be added.
562     */
563     void dequeueNextAgentEvents(xxxx::EventContainer& events);
564
565     /** Convenience method for debugging/troubleshooting.
566
567         \return The highest timestamp from the events in the bottom.
568         If no events are present this method returns TIME_INFINITY.
569     */
570     xxxx::Time maxTime() const {
571         // purely for debugging
572         return (!empty() ? front()->getReceiveTime() : TIME_INFINITY);
573     }
574
575     /** Convenience method for debugging/troubleshooting.
576
577         \return The minimum timestamp from the events in the bottom.
578         If no events are present this method returns TIME_INFINITY.
579     */
580     xxxx::Time findMinTime() const {
581         // purely for debugging
582         return (!empty() ? back()->getReceiveTime() : TIME_INFINITY);
583     }
584
585     /** Method to determine the range of receive time values currently
586         in bottom.  This value is typically used to decide if it is
587         worth moving events from bottom into the ladder.
588
589         \return The difference in maximum and minimum receive
590         timestamp of events in the bottom.  This value is zero if all
591         events have the same receive time.  If the bottom is empty,
592         then this method also returns zero.
593     */
594     xxxx::Time getTimeRange() const {
595         if (empty()) {
596             return 0;
597         }
598         return (front()->getReceiveTime() - back()->getReceiveTime());
599     }
600
```

```
601     /** Determine bucket width to move bottom into ladder.
602
603         This method is invoked only when the ladder is empty and the
604         bottom is long and needs to be moved into the ladder.  This
605         method must compute and return the preferred bucket width.
606
607         \note If the bottom is empty this method returns bucket width
608         of 0.
609      */
610     double getBucketWidth() const;
611
612     /** Convenience method to check if the entries in the bottom are
613         sorted correctly.  This method is purely used for
614         troubleshooting/debugging.
615     */
616     void validate() const;
617
618     /** Event comparison function used by various structures in ladder
619         queue.
620
621         \param[in] lhs The left-hand-side event for comparison.  The
622         pointer cannot be NULL.
623
624         \param[in] lhs The right-hand-side event for comparison.  The
625         pointer cannot be NULL.
626
627         \return This method returns true if lhs is less than rhs.
628         That is, lhs should be scheduled before rhs.
629     */
630     static inline bool compare(const xxxx::Event* const lhs,
631                                const xxxx::Event* const rhs) {
632         return ((lhs->getReceiveTime() > rhs->getReceiveTime()) ||
633                 ((lhs->getReceiveTime() == rhs->getReceiveTime() &&
634                   (lhs->getReceiverAgentID() > rhs->getReceiverAgentID()))));
635     }
636
637     /** Convenience method to check to see if bottom has events before
638         the specified receive time.
639
640         This method is used for troubleshooting/debugging only.
641
642         \param[in] recvTime The receive time for checking.
643
644         \return Returns true if an event before this receiveTime (for
645         any agent) is pending in the bottom rung.
646     */
647     bool haveBefore(const Time recvTime) const {
648         return (findMinTime() <= recvTime);
649     }
650
651     /** Convenience method to consistently access the first event in
652         the bottom, consistent with the way bottom is sorted.
653
654         \note Calling this method when the bottom is empty has
655         undefined behavior.
656
657         \return The next event with the lowest time stamp.
658     */
659     xxxx::Event* first_event() {
660         return back();
661     }
662
663 protected:
664     // Currently this class does not have any protected members.
665
666 private:
667     // Currently this class does not have any private members
668 };
669
670 /** Class that represents one rung in the 2-tier ladder queue.
671
672     The 2-tier rung uses the same strategy for receive time-based
673     bucket creation as the regular ladder queue.  However, the
674     organization of each bucket is different -- events are not stored
675     in a linear list.  Instead they are stored in sub-buckets based on
```



```
676     a hash of the sender agent's ID.
677 */
678 class TwoTierRung {
679 public:
680     /** The constructor to create an empty rung.
681
682         The constructor merely initializes all the instance variables
683         to default initial values to create an empty rung.
684     */
685     TwoTierRung() : rStartTS(TIME_INFINITY), rCurrTS(TIME_INFINITY),
686                     bucketWidth(0), currBucket(0), rungEventCount(0) {
687         LQ2T_STATS(maxBkts = 0);
688     }
689
690     /** A move constructor required to quickly move rungs in a ladder
691         to shrink/grow it.
692
693         \param[in] src The source rung from where events are to be
694         copied.
695     */
696     TwoTierRung(TwoTierRung&& src) :
697         rStartTS(src.rStartTS), rCurrTS(src.rCurrTS),
698         bucketWidth(src.bucketWidth), currBucket(src.currBucket),
699         bucketList(std::move(src.bucketList)),
700         rungEventCount(src.rungEventCount) {
701         LQ2T_STATS(maxBkts = src.maxBkts);
702     }
703
704     /** Convenience constructor to create a rung using events from the
705         top rung.
706
707         This is a delegating constructor that delegates the actual
708         tasks to the overloaded constructor.
709
710         \param[in] top The top bucket from where the events are to be
711         created.
712     */
713     explicit TwoTierRung(TwoTierTop&& top) :
714         TwoTierRung(std::move(top), top.getMinTime(),
715                     top.getBucketWidth()) {
716             // Reset of top counters etc. is done by caller in
717             // TwoTierLadderQueue::populateBottom()
718     }
719
720     /** Convenience constructor to create a rung with events from a
721         given bucket.
722
723         \param[in,out] bkt The bucket from where events are to be
724         moved into this newly created rung.  After this operation data
725         in the bucket is cleared.
726
727         \param[in] rStart The start time for this rung.
728
729         \param[in] bucketWidth The delta in receive time for each
730         bucket in this rung.  The bucketWidth must be > 0.
731     */
732     TwoTierRung(TwoTierBucket&& bkt, const Time rStart,
733                 const double bucketWidth) : rungEventCount(0) {
734         move(std::move(bkt), rStart, bucketWidth);
735     }
736
737     /** Convenience method initialize a rung by moving events from a
738         given bucket.
739
740         It is assumed that this rung is empty prior to this operation.
741
742         \param[in,out] bkt The bucket from where events are to be
743         moved into this rung.  After this operation data in the bucket
744         is cleared.
745
746         \param[in] rStart The start time for this rung.
747
748         \param[in] bucketWidth The delta in receive time for each
749         bucket in this rung.  The bucketWidth must be > 0.
750     */
```

```
751     void move(TwoTierBucket&& bucket, const Time rStart,
752               const double bucketWidth);
753
754     /** Convenience constructor to create a rung with events from the
755         bottom rung.
756
757         This operation is used to redistribute bottom to the ladder
758         ensures that the bottom does not get too long.
759
760         \param[in,out] bottom The bottom rung from where events are to
761         be moved into this newly created rung.  After this operation
762         bottom will be empty.
763
764         \param[in] rStart The start time for this rung.
765
766         \param[in] bucketWidth The delta in receive time for each
767         bucket in this rung.  The bucketWidth must be > 0.
768     */
769     TwoTierRung(OneTierBottom&& bottom, const Time rStart,
770                 const double bucketWidth) : rungEventCount(0) {
771         move(std::move(bottom), rStart, bucketWidth);
772     }
773
774     /** Convenience method to create a rung with events from the
775         bottom rung.
776
777         This operation is used to redistribute bottom to the ladder
778         ensures that the bottom does not get too long.  This method
779         assumes that the this rung is empty to begin with.
780
781         \param[in,out] bottom The bottom rung from where events are to
782         be moved into this newly created rung.  After this operation
783         bottom will be empty.
784
785         \param[in] rStart The start time for this rung.
786
787         \param[in] bucketWidth The delta in receive time for each
788         bucket in this rung.  The bucketWidth must be > 0.
789     */
790     void move(OneTierBottom&& bottom, const Time rStart,
791               const double bucketWidth);
792
793     /** Remove the next bucket in this rung for moving to another rung
794         in the ladder.
795
796         This method must be used to remove the next bucket from this
797         rung.  The bucket is logically removed (or moved) out of this
798         rung.
799
800         \param[out] bktTime The simulation receive time associated
801         with the bucket being moved out.
802     */
803     TwoTierBucket&& removeNextBucket(xxxx::Time& bktTime);
804
805     /** Determine if this rung is empty.
806
807         This is a convenience method that is used to determine if this
808         rung contains any events to be processed.
809
810         \return This method returns true if the rung does not have any
811         events -- i.e., when the rung is empty.
812     */
813     bool empty() const { return (rungEventCount == 0); }
814
815     /** Add an event to suitable bucket in this rung.
816
817         This method computes a bucket index (based on equation #2 in
818         LQ paper) using the formula:
819
820         \code
821         size_t bucketNum = (event->getReceiveTime() - rStartTS) / bucketWidth;
822         \endcode
823
824         \param[in] event The event to be added to a suitable bucket in
825         this rung.
```



```
826     */
827     void enqueue(xxxx::Event* event);
828
829     /** Obtain the start time for this rung.
830
831         This method returns the rung starting time that was set when
832         this rung was created.
833
834         \return The starting time of this rung that determines the
835         lowest timestamp event that can be added to this rung.
836     */
837     xxxx::Time getStartTime() const { return rStartTS; }
838
839     /** Obtain the bucket width (i.e., difference in receive times for
840         adjacent buckets) for this rung.
841
842         This method returns the bucket width that was set when this
843         rung was created.
844
845         \return The bucket with for this rung.
846     */
847     double getBucketWidth() const { return bucketWidth; }
848
849     /** The current bucket value in this ladder queue.
850
851         The current minimum time of events that can be added to this
852         rung of the ladder eueue.
853
854         \return The minimum timestamp of events that can be added to
855         the rung of this ladder queue.
856     */
857     xxxx::Time getCurrTime() const {
858         return rCurrTS;
859     }
860
861     /** The maximum receive time value of event that can be added to
862         this rung.
863
864         \return The maximum receive time of an event that can be added
865         to a bucket in this rung.
866     */
867     xxxx::Time getMaxRungTime() const {
868         return rStartTS + (bucketList.size() * bucketWidth);
869     }
870
871     /** Convenience method to determine if a given event can be added
872         to this rung.
873
874         \param[in] event The event whose receive time is to be used to
875         check to see if it can be added to this ladder.
876
877         \return Returns true if the event can be added to this rung.
878         Otherwise it returns false.
879     */
880     bool canContain(xxxx::Event* event) const;
881
882     /** Remove all events from the given sender sent at-or-after the
883         specified send time from all buckets in this rung.
884
885         This method linearly scans the buckets, checks, and removes
886         all events that were sent by the sender at-or-after the
887         specified send time.
888
889         \param[in] sender The sender agent whose events are to be
890         removed.
891
892         \param[in] sendTime The time at-or-after which events from the
893         sender are to be removed from the given list.
894
895         \param[out] ceScanRung The stats object to be updated with
896         number of events scanned in the buckets in this rung.
897
898         \return This method returns the total number of events that
899         were removed from this rung.
900     */
```

```
901     int remove_after(xxxx::AgentID sender, const Time sendTime
902                      LQ2T_STATS(COMMA Avg& ceScanRung));
903
904     /** Remove all events for a given receiver agent in this rung.
905
906         This is a convenience method that removes all events for a
907         given receiver agent in this rung.  This method is used to
908         remove events scheduled for an agent, when an agent is removed
909         from the scheduler.
910
911         \param[in] receiver The receiving agent ID whose events are to
912         be removed from all the buckets in this rugn.
913
914         \param[out] ceScanRung The stats object to be updated with
915         number of events scanned in the buckets in this rung.
916     */
917     int remove(xxxx::AgentID receiver
918                LQ2T_STATS(COMMA Avg& ceScanRung));
919
920     /** Check to ensure that the number of events in various buckets
921         matches the count instance variable.
922
923         This method is used only for troubleshooting/debugging
924         purposes.  If counts don't match then assert fails in this
925         method causing the simulation to abort.
926     */
927     void validateEventCounts() const;
928
929     /** Print a user-friendly version of the events in this queue.
930
931         Currently this method is not implemented.
932     */
933     void prettyPrint(std::ostream& os) const;
934
935     /** Update the statistics object with data from this rung.
936
937         \param[out] avgBktCnt Update the average number of buckets in
938         this rung.
939     */
940     void updateStats(Avg& avgBktCnt) const;
941
942     /** Convenience method to determine if the current bucket in this
943         rung is empty.
944
945         \return This method returns true if the current bucket in this
946         rung is empty.
947     */
948     bool isCurrBucketEmpty() const {
949         return (currBucket >= bucketList.size() ||
950                 bucketList[currBucket].empty());
951     }
952
953     /** This method is purely for troubleshooting one scenario where
954         an event would get stuck in the ladder and not get scheduled
955         correctly.
956
957         \param[in] recvTime The time to be used for checking to see if
958         sub-buckets have an event before this time.
959
960         \return Returns true if an event before this receiveTime (for
961         any agent) is pending in a sub-bucket.
962     */
963     bool haveBefore(const Time recvTime) const;
964
965 protected:
966     // Currently this class does not have any protected members.
967
968 private:
969     /** The lowest timestamp event that can be added to this rung.
970         This value is set when a rung is created and is never changed
971         during the lifetime of this rung.
972     */
973     xxxx::Time rStartTS;
974
975     /** The timestamp of the lowest event that can be currently added
```



```
976         to this rung.  This value logically starts with rStartTS and
977         grows to the time stamp of last bucket in this rung as buckets
978         are dequeued from this rung.
979     */
980     xxxx::Time rCurrTS;
981
982     /** The width of the bucket in simulation receive time
983         differences.  This value can be fractional.
984     */
985     double bucketWidth;
986
987     /** The index of the current bucket on this rung to which events
988         can be added.  This is also the next bucket that will be
989         dequeued from the rung.
990     */
991     size_t currBucket;
992
993     /** The deque containing the set of vectors in this bucket list.
994      */
995     std::deque<TwoTierBucket> bucketList;
996
997     /** Total number of events still present in this rung.  This is
998         used to report size and check for empty quickly.
999     */
1000    int rungEventCount;
1001
1002    /** Statistics object to track the maximum number of buckets used
1003        in this rung */
1004    LQ2T_STATS(size_t maxBkts);
1005 };
1006
1007 /** The top-level 2-tier ladder queue
1008
1009     <p>This class represents the top-level 2-tier ladder queue class
1010     that interfaces with the XXXX scheduler.  This class implements
1011     the top-level logic associated with ladder queue to enqueue,
1012     dequeue, and cancel events from the ladder queue.</p>
1013
1014     <p>The logic for most of the operations is consistent with those
1015     proposed by the Tang et. al, except for the following:
1016
1017     <ol>
1018
1019     <li>The size of the bottom is not restricted.  So events are never
1020     moved from bottom back into the ladder.</li>
1021
1022     <li> The number of buckets in a rung is restricted to 100</li>
1023
1024     </ol>
1025
1026     </p>
1027 */
1028 class TwoTierLadderQueue : public EventQueue {
1029 public:
1030     /** The constructor that creates an empty ladder queue.
1031
1032         The constructor also initializes various statistics variables
1033         used by the this queue to report detailed statistics about its
1034         operations at the end of simulation.
1035      */
1036     TwoTierLadderQueue() : EventQueue("LadderQueue"), nRung(0),
1037                            ladderEventCount(0) {
1038         ladder.reserve(MaxRungs);
1039         LQ2T_STATS(ceTop    = ceLadder   = ceBot  = 0);
1040         LQ2T_STATS(insTop   = insLadder  = insBot = 0);
1041         LQ2T_STATS(maxRungs = maxBotSize = 0);
1042     }
1043
1044     /** The destructor.
1045
1046         Currently the destructor does not have anything special to do
1047         as the different encapsulated objects handle all the necessary
1048         clean-up.
1049     */
1050     ~TwoTierLadderQueue() {}
```

```
1051
1052        /** Enqueue an event into the laadder queue.
1053
1054            Depending on the scenario the event is appropriately added to
1055            one of: top, ladder rung, or the bottom.
1056
1057            \param[in] e The event to be enqueued for scheduling in the
1058            ladder queue.
1059        */
1060        void enqueue(xxxx::Event* e);
1061
1062        /** Cancel all events from a given sender that were sent
1063            at-or-after the specified send time.
1064
1065            This method essentially calls the corresponding method(s) in
1066            top, rung, and bottom to cancel pending events.
1067
1068            \param[in] sender The sender agent whose events are to be
1069            removed.
1070
1071            \param[in] sendTime The time at-or-after which events from the
1072            sender are to be removed from the given list.
1073
1074            \return This method returns the number of events that were
1075            removed.
1076        */
1077        int remove_after(xxxx::AgentID sender, const Time sendTime);
1078
1079        /** Determine if the ladder queue is empty.
1080
1081            Implements the interface method used by XXXX::Scheduler.
1082
1083            \return Returns true if top, ladder, and bottom are all empty
1084            -- i.e., there are no pending events.
1085        */
1086        virtual bool empty() {
1087            return top.empty() && (ladderEventCount == 0) &&  bottom.empty();
1088        }
1089
1090        /** Implementation for method used by XXXX::Scheduler.
1091
1092            This method is called by XXXX kernel to inform the scheduler
1093            queue about an agent being added during initialization.  The
1094            ladder queue does not utilize this information and
1095            consequently this method does not have any special operation
1096            to perform.
1097
1098            \param[in] agent The agent being added.  This pointer is not
1099            really used.
1100
1101            \return This method simply returns nullptr as the ladder queue
1102            does not use any cross references in xxxx::Agent for its
1103            operations.
1104        */
1105        virtual void* addAgent(xxxx::Agent* agent);
1106
1107        /** Remove an agent just before simulation completes.
1108
1109            This method is invoked by the XXXX kernel to inform that an
1110            agent is being removed.  This method removes all pending
1111            events for the specified agent from the ladder queue.
1112
1113            \param[in] agent The agent whose sender ID is used to remove
1114            all pending events in the top, rungs, and bottom.
1115        */
1116        virtual void removeAgent(xxxx::Agent* agent);
1117
1118        /** Implement interface method to peek at the next event to
1119            schedule.
1120
1121            \note In order to enable peeking of the front event, the
1122            bottom may need to get populated.
1123
1124            \return A pointer to the next event to schedule (if any).  The
1125            event is not dequeued.
```



```
1126         */
1127        virtual xxxx::Event* front();
1128
1129        /** This method is used to provide necessary implemntation to
1130            interface with the XXXX scheduler.  This method dequeues the
1131            next batch of the concurrent events for processing by a given
1132            agent.
1133
1134            \param[out] events The container to which all the events to be
1135            processed is to be added.
1136        */
1137        virtual void dequeueNextAgentEvents(xxxx::EventContainer& events);
1138
1139        /** Add an event to be scheduled to this ladder queue.
1140
1141            This method implements the core API used by agents to schedule
1142            events for each other.
1143
1144            \param[in] agent The receiver agent for which the event is
1145            scheduled.  This pointer is not used.
1146
1147            \param[in] event The event to be scheduled. This simply calls
1148            the overloaded enqueue method.  The reference count on the
1149            event is increased by this method to account for this event
1150            being present in the ladder queue.
1151        */
1152        virtual void enqueue(xxxx::Agent* agent, xxxx::Event* event);
1153
1154        /** Enqueue a batch of events
1155
1156            This API to schedule a block of events.  This API is typically
1157            used after a rollback.
1158
1159            \param[in] agent The receiver agent for which the event is
1160            scheduled.  This pointer is not used.
1161
1162            \param[in] events The list of events to be scheduled. This
1163            simply calls the overloaded enqueue method to enqueue one
1164            event at a time.
1165        */
1166        virtual void enqueue(xxxx::Agent* agent, xxxx::EventContainer& events);
1167
1168        /** Implement XXXX kernel API to cancel all events sent by a given
1169            agent after a given time.
1170
1171            \param[in] dest The destination agent whose events are to be
1172            cancelled.
1173
1174            \param[in] sender The sender agent ID whose events are to be
1175            cancelled.
1176
1177            \param[in] sentTime The send time at-or-after which all events
1178            from the sender are to be cancelled.
1179        */
1180        virtual int eraseAfter(xxxx::Agent* dest, const xxxx::AgentID sender,
1181                               const xxxx::Time sentTime);
1182
1183        /** Print a human understandable version of the events in this
1184            queue.
1185
1186            currently this method is not implemented.
1187         */
1188        virtual void prettyPrint(std::ostream& os) const;
1189
1190
1191        /** Convenience method to check to see if ladder queue has events
1192            before the specified receive time.
1193
1194            This method is used for troubleshooting/debugging only.
1195
1196            \param[in] recvTime The receive time for checking.
1197
1198            \return Returns true if an event before this receiveTime (for
1199            any agent) is pending in the bottom rung.
1200        */
```

```
1201      bool haveBefore(const Time recvTime,
1202                      const bool checkBottom = false) const;
1203
1204      /** Method to report aggregate statistics.
1205
1206          This method is invoked at the end of simulation after all
1207          agents on this rank have been finalized.  This method is meant
1208          to report any aggregate statistics from this queue.  This
1209          method writes statistics only if LQ2T_STATS macro is enabled.
1210
1211          \param[out] os The output stream to which the statistics are
1212          to be written.
1213      */
1214      virtual void reportStats(std::ostream& os);
1215
1216      /** The maximum number of rungs that are normally created in the
1217          ladder queue.  The default value for this set to 8 based on
1218          the value suggested by Tang et. al. in the original Ladder
1219          Queue paper.  However, this value can make some difference in
1220          the overall performance and possibly fine tuned to suit the
1221          application needs based on the concurrency and number of
1222          events in the model.
1223      */
1224      static size_t MaxRungs;
1225
1226  protected:
1227      /** Check and create rungs in the ladder and return the next
1228          bucket of events from the ladder.
1229
1230          This method implements the corresponding recurseRung method
1231          from the LQ paper. Refer to the paper for the details.
1232      */
1233      TwoTierBucket&& recurseRung();
1234
1235      /** This is a convenience method that is used to move events from
1236          the ladder into bottom.
1237
1238          This method moves events from teh current bucket in the last
1239          rung of the ladder into the bottom.  If this method is called
1240          when bottom is not empty, it does not perform any operation
1241          and returns immediately.  If the ladder does not have any
1242          events, but the top has events, then this method first moves
1243          events from top-rung into the ladder and then removes events
1244          from the last rung into the bottom-rung.
1245      */
1246      void populateBottom();
1247
1248      /** Method to create a new ladder rung from the current bottom.
1249
1250          This method should be called only when the following 2
1251          conditions are met:
1252
1253          1. Length of bottom is > LQ2T_THRESH
1254
1255          2. The bottom has events that are at different time stamps --
1256             that is bottom.getTimeRange() > 0.
1257
1258          \return This method returns the index of the rung created so
1259          that the caller can readily work with that rung.
1260      */
1261      int createRungFromBottom();
1262
1263  private:
1264      TwoTierTop top;
1265
1266      /** The ladder in the queue.  The lader consists of a set of
1267          rungs.  The currently used rung in the ladder is indicated by
1268          the nRung instance variable.  If the ladder is empty, then
1269          nRung is (or should be) 0
1270      */
1271      std::vector<TwoTierRung> ladder;
1272
1273      /** The currently used last rung in the ladder queue.  If the
1274          ladder is empty, then nRung is (or should be) 0.  Otherwise
1275          this value is (or should be) in the range 0 < nRung <=
```



```
1276          ladder.size().  Rungs below nRung are not used and they do not
1277          contain any events to be scheduled.
1278      */
1279      size_t nRung;
1280
1281      /** Instance variable to track the current number of pending
1282          events in all the rungs of the ladder.  This is a convenience
1283          instance variable to quickly detect pending events in the
1284          ladder without having to iterate through each rung.
1285      */
1286      int ladderEventCount;
1287
1288      OneTierBottom bottom;
1289
1290      LQ2T_STATS(Avg ceTop);
1291      LQ2T_STATS(Avg ceBot);
1292      LQ2T_STATS(Avg ceLadder);
1293
1294      LQ2T_STATS(Avg ceScanTop);
1295      LQ2T_STATS(Avg ceScanLadder);
1296
1297      /** The ceScanBot statistic tracks size of bottom rung scanned
1298          when at one (or more) events were canceled from bottom.
1299      */
1300      LQ2T_STATS(Avg ceScanBot);
1301
1302      /** The ceNoCanScanBot statistic tracks size of bottom rung
1303          scanned but did not cancel any events.
1304      */
1305      LQ2T_STATS(Avg ceNoCanScanBot);
1306
1307      LQ2T_STATS(int insTop);
1308      LQ2T_STATS(int insLadder);
1309      LQ2T_STATS(int insBot);
1310      LQ2T_STATS(size_t maxRungs);
1311      LQ2T_STATS(Avg avgBktCnt);
1312      LQ2T_STATS(Avg botLen);
1313      LQ2T_STATS(Avg avgBktWidth);
1314
1315      /** Gague to track the number of events and times bottom was
1316          redistributed to the last rung of the ladder.
1317
1318          Redistributing bottom to the ladder ensures that the bottom
1319          does not get too long.  But it is an expensive operation
1320          because all the sorting that was done is lost.  So it is a
1321          balance and we track and report this number for reference.
1322      */
1323      LQ2T_STATS(Avg botToRung);
1324
1325      /** Gauge to track the maximum length of bottom.  The length of
1326          bottom plays an important role in the overall performance of
1327          the ladder queue.
1328      */
1329      LQ2T_STATS(size_t maxBotSize);
1330  };
1331
1332
1333  END_NAMESPACE(xxxx)
1334
1335  #endif
```

```
#ifndef TWO_TIER_LADDER_QUEUE_CPP
#define TWO_TIER_LADDER_QUEUE_CPP

#include <algorithm>
#include <functional>
#include "TwoTierLadderQueue.h"

// The maximum number of buckets 1 rung can have.
#define MAX_BUCKETS 100

/** The number of sub-buckets to be used in each 2-tier bucket */
int xxxx::TwoTierBucket::t2k = 32;

// ----------------------[ TwoTierBucket methods ]----------------------

xxxx::TwoTierBucket::~TwoTierBucket() {
    clear();
}

void
xxxx::TwoTierBucket::clear() {
    for (BktEventList& subBkt : subBuckets) {
        for (xxxx::Event* event : subBkt) {
            event->decreaseReference();
        }
    }
    count = 0;
}

void
xxxx::TwoTierBucket::push_back(TwoTierBucket&& srcBkt) {
    ASSERT(srcBkt.subBuckets.size() == (size_t) t2k);
    ASSERT(srcBkt.subBuckets.size() == subBuckets.size());
    // Move evens from srcBkt into corresponding sub-buckets
    for (int idx = 0; (idx < t2k); idx++) {
        // Obtain reference to subbucket to be moved.
        BktEventList& src  = srcBkt.subBuckets[idx];
        BktEventList& dest = subBuckets[idx];
        // Move events from src to dest.
        dest.insert(dest.end(), src.begin(), src.end());
        // Update counters (also used to troubleshooting).
        count += src.size();
        // Clear out the source as events have been logically moved
        // out of it.
        src.clear();
    }
}

void
xxxx::TwoTierBucket::push_back(BktEventList& dest, TwoTierBucket&& srcBkt) {
    // Move all entries from each sub-bucket in srcBkt to the end of dest.
    for (BktEventList& subBkt : srcBkt.subBuckets) {
        dest.insert(dest.end(), subBkt.begin(), subBkt.end());
        subBkt.clear();
    }
    // Reset count as part of move semantics
    srcBkt.count = 0;
}

int
xxxx::TwoTierBucket::remove_after(xxxx::AgentID sender, const Time sendTime
                                  LQ2T_STATS(COMMA Avg& scans)) {
    const size_t subBktIdx = hash(sender);
    LQ2T_STATS(scans += subBuckets[subBktIdx].size());
    int removedCount = remove_after(subBuckets[subBktIdx], sender, sendTime);
    count -= removedCount;  // Track remaining events
    return removedCount;
}

// Helper method to remove events from a sub-bucket.
int
xxxx::TwoTierBucket::remove_after(BktEventList& list, xxxx::AgentID sender,
                                  const Time sendTime) {
    size_t removedCount = 0;
    size_t curr = 0;
```



```
    while (curr < list.size()) {
        xxxx::Event* const event = list[curr];
        if ((event->getSenderAgentID() == sender) &&
            (event->getSentTime() >= sendTime)) {
            // Free-up event.
            event->decreaseReference();
            removedCount++;
            // To minimize removal time replace entry with last one
            // and pop the last entry off.
            list[curr] = list.back();
            list.pop_back();
        } else {
            curr++;  // on to the next event in the list
        }
    }
    return removedCount;
}

// This method is not performance critical as it is only called once
// at the end of simulation.
int
xxxx::TwoTierBucket::remove(xxxx::AgentID receiver) {
    size_t removedCount = 0;
    // Remove events from each sub-bucket.
    for (BktEventList& list : subBuckets) {
        // Use helper method to remove events.
        removedCount += remove(list, receiver);
    }
    count -= removedCount;  // Track remaining events
    return removedCount;
}

// static helper method also used by OneTierBottom.  This is called
// few times at the end of simulation.  So it is not performance
// critical.
int
xxxx::TwoTierBucket::remove(BktEventList& list, xxxx::AgentID receiver) {
    int removedCount = 0;  // statistics tracking
    // Linear scan through events in a given sub-bucket
    BktEventList::iterator curr = list.begin();
    while (curr != list.end()) {
        if ((*curr)->getReceiverAgentID() == receiver) {
            (*curr)->decreaseReference();
            curr = list.erase(curr);
            removedCount++;
        } else {
            curr++;
        }
    }
    return removedCount;  // let caller know the events removed
}

// This method is not performance critical.  It is used only for
// troubleshooting/debugging
bool
xxxx::TwoTierBucket::haveBefore(const Time recvTime) const {
    for (const BktEventList& list : subBuckets) {
        for (const xxxx::Event* const event : list) {
            if (event->getReceiveTime() <= recvTime) {
                return true;
            }
        }
    }
    return false;
}

// Actually counts events in each bucket for validation purposes.
// This method is not performance critical.  It is used only for
// troubleshooting/debugging
size_t
xxxx::TwoTierBucket::getEventCount() const {
    int sum = 0;
    for (const BktEventList& subBkt : subBuckets) {
        sum += subBkt.size();
    }
```

```
151     return sum;  // total number of events
152 }
153
154 // -------------------------[ TwoTierTop methods ]-------------------------
155
156 // Helper method called from constructor and when events are moved
157 // from top into ladder.
158 void
159 xxxx::TwoTierTop::reset(const Time startTime) {
160     minTS    = TIME_INFINITY;
161     maxTS    = 0;
162     topStart = startTime;
163     clear();
164 }
165
166 void
167 xxxx::TwoTierTop::add(xxxx::Event* event) {
168     push_back<SenderID>(event);   // Call base-class method.
169     // Update running timestamps.
170     minTS = std::min(minTS, event->getReceiveTime());
171     maxTS = std::max(maxTS, event->getReceiveTime());
172 }
173
174 // -------------------------[ OneTierBottom methods ]-------------------------
175
176 void
177 xxxx::OneTierBottom::enqueue(xxxx::TwoTierBucket&& bucket) {
178     // Move events from bucket into the bottom.
179     TwoTierBucket::push_back(*this, std::move(bucket));
180     // Now sort the whole bottom O(n*log(n)) operation
181     std::sort(begin(), end(), OneTierBottom::compare);
182     DEBUG(validate());
183 }
184
185 void
186 xxxx::OneTierBottom::enqueue(xxxx::Event* event) {
187     BktEventList::iterator iter =
188         std::lower_bound(begin(), end(), event, compare);
189     insert(iter, event);  // base class method.
190     DEBUG(validate());
191 }
192
193 void
194 xxxx::OneTierBottom::dequeueNextAgentEvents(xxxx::EventContainer& events) {
195     if (empty()) {
196         return;  // no events to provide
197     }
198     // Reference information used for checking in the loop below.
199     const xxxx::Event*  nextEvt  = back();
200     const xxxx::AgentID receiver = nextEvt->getReceiverAgentID();
201     const xxxx::Time    currTime = nextEvt->getReceiveTime();
202     // Move all events from bottom to the events-container for scheduling.
203     do {
204         // Back event is the lowest timestamp (or highest priority)
205         // based on sorting order in OneTierBottom::compare()
206         xxxx::Event* event = back();
207         events.push_back(event);
208         pop_back();   // remove from bottom.
209         // erase(begin());
210         // Check and work with the next event.
211         nextEvt = (!empty() ? back() : NULL);
212         DEBUG(std::cout << "Delivering: " << *event << std::endl);
213     } while (!empty() && (nextEvt->getReceiverAgentID() == receiver) &&
214              TIME_EQUALS(nextEvt->getReceiveTime(), currTime));
215     DEBUG(validate());
216 }
217
218 double
219 xxxx::OneTierBottom::getBucketWidth() const {
220     if (empty()) {
221         return 0;
222     }
223     ASSERT(front() != NULL);
224     ASSERT(back()  != NULL);
225     // Assumes that bottom is sorted with the lowest timestamp at the
```



```
226     // end for fast pop_back
227     const double maxTS = front()->getReceiveTime();
228     const double minTS = back()->getReceiveTime();
229     return (maxTS - minTS + size() - 1.0) / size();
230 }
231
232 int
233 xxxx::OneTierBottom::remove_after(xxxx::AgentID sender, const Time sendTime) {
234     // Since bucket is sorted we can shortcircuit scan if last event's
235     // time is less-or-equal to sendTime.
236     if (empty() || (sendTime >= front()->getReceiveTime())) {
237         return -1;  // Since bucket does not have events to be cancelled.
238     }
239     size_t removedCount = 0;
240     iterator curr = begin();
241     while (curr != end()) {
242         xxxx::Event* const event = *curr;
243         if ((event->getSenderAgentID() == sender) &&
244             (event->getSentTime() >= sendTime)) {
245             // Free-up event.
246             event->decreaseReference();
247             removedCount++;
248             // In sorted mode we have to preserve the order. So
249             // cannot swap & pop in this situation
250             curr = erase(curr);
251         } else {
252             curr++;  // onto next event
253         }
254     }
255     return removedCount;
256 }
257
258 // This method is used only for debugging. So it is not performance
259 // critical.
260 void
261 xxxx::OneTierBottom::validate() const {
262     if (empty()) {
263         return;  // yes. bottom is valid.
264     }
265     // Ensure events are sorted in timestamp order.
266     BktEventList::const_iterator next = cbegin();
267     BktEventList::const_iterator prev = next++;
268     while ((next != cend()) &&
269            ((*next)->getReceiveTime() >= (*prev)->getReceiveTime())) {
270         prev = next++;
271     }
272     if (next != cend()) {
273         std::cout << "Error in LadderQueue.Bottom: Event " << **next
274                   << " was found after " << **prev << std::endl;
275     }
276     ASSERT( next == cend() );
277 }
278
279 // -------------------------[ TwoTierRung methods ]-------------------------
280
281 void
282 xxxx::TwoTierRung::move(TwoTierBucket&& bkt, const Time minTS,
283                         const double bktWidth) {
284     // Setup starting & current timestamp for this rung.
285     rStartTS = rCurrTS = minTS;
286     // Ensure bucket width is not ridiculously small
287     bucketWidth = bktWidth;
288     currBucket  = 0;  // current bucket in this rung.
289     // Initialize variable to track maximum bucket count
290     LQ2T_STATS(maxBkts = 0);
291     DEBUG(std::cout << "bucketWidth = " << bucketWidth << std::endl);
292     ASSERT(bucketWidth > 0);
293     ASSERT(rungEventCount == 0);
294     // Move events from given bucket into buckets in this Rung.
295     DEBUG(std::cout << "Adding " << bkt.size() << " events to rung\n");
296     for (BktEventList& list : bkt.getSubBuckets()) {
297         // Add all events from sub-buckets to various buckets in this rung.
298         while (!list.empty()) {
299             // Remove event from the top linked list.
300             xxxx::Event* event = list.back();
```

```
            list.pop_back();
            // Add to the appropriate bucket in this rung using a
            // helper method in this class.
            enqueue(event);
        }
    }
    // Reset bucket counters as we have moved all the events out
    bkt.resetCount();
    DEBUG(validateEventCounts());
}

void
xxxx::TwoTierRung::move(OneTierBottom&& bottom, const Time rStart,
                        const double bktWidth) {
    rStartTS = rCurrTS = rStart;
    // Ensure bucket width is not ridiculously small
    bucketWidth = bktWidth;
    currBucket  = 0;  // current bucket in this rung.
    ASSERT(rungEventCount == 0);
    // Initialize variable to track maximum bucket count
    LQ2T_STATS(maxBkts = 0);
    DEBUG(std::cout << "bucketWidth = " << bucketWidth << std::endl);
    ASSERT(bucketWidth > 0);
    ASSERT(rungEventCount == 0);
    // Move events from bottom into buckets in this Rung.
    DEBUG(std::cout << "Adding " << bottom.size() << " events to rung\n");
    for (xxxx::Event* event : bottom) {
        // Add to the appropriate bucket in this rung using a
        // helper method in this class.
        enqueue(event);
    }
    // Finally clear out the events in bottom.
    bottom.clear();
    DEBUG(validateEventCounts());
}


bool
xxxx::TwoTierRung::canContain(xxxx::Event* event) const {
    const xxxx::Time recvTime = event->getReceiveTime();
    const int bucketNum = (recvTime - rStartTS) / bucketWidth;
    return ((bucketNum >= (int) currBucket) && (recvTime >= rStartTS));
}

void
xxxx::TwoTierRung::enqueue(xxxx::Event* event) {
    ASSERT(event != NULL);
    ASSERT(event->getReceiveTime() >= getCurrTime());
    // Compute bucket for this event based on equation #2 in LQ paper.
    size_t bucketNum = (event->getReceiveTime() - rStartTS) / bucketWidth;
    ASSERT(bucketNum >= currBucket);
    if (bucketNum >= bucketList.size()) {
        // Ensure bucket list of sufficient size
        bucketList.resize(bucketNum + 1);
        // update variable to track maximum bucket count
        LQ2T_STATS(maxBkts = std::max(maxBkts, bucketList.size()));
    }
    ASSERT(bucketNum < bucketList.size());
    // Add event into appropriate bucket
    bucketList[bucketNum].push_back<SenderID>(event);
    // Track number of events added to this Rung
    rungEventCount++;
}

xxxx::TwoTierBucket&&
xxxx::TwoTierRung::removeNextBucket(xxxx::Time& bktTime) {
    ASSERT(!empty());
    ASSERT(currBucket < bucketList.size());
    // Find next non-empty bucket in this rung (there has to be one as
    // the previous asserts passed necessary checks)
    while ((currBucket < bucketList.size()) && bucketList[currBucket].empty()) {
        currBucket++;
    }
    DEBUG(validateEventCounts());
    ASSERT(currBucket < bucketList.size());
```



```
    ASSERT(!bucketList[currBucket].empty());
    // Track events that will be removed when this method returns
    rungEventCount -= bucketList[currBucket].size();
    ASSERT(rungEventCount >= 0);
    // Save information about the bucket to be removed & returned.
    const int retBkt = currBucket;
    bktTime = rStartTS + (retBkt * bucketWidth);
    // Advance current bucket to next time.
    currBucket++;
    rCurrTS = rStartTS + (currBucket * bucketWidth);
    // Sanity check on counters...
    if (currBucket >= bucketList.size()) {
        ASSERT(rungEventCount == 0);
    }
    return std::move(bucketList[retBkt]);
}

int
xxxx::TwoTierRung::remove_after(xxxx::AgentID sender, const Time sendTime
                                LQ2T_STATS(COMMA Avg& ceScanRung)) {
    if (empty() || (sendTime > getMaxRungTime())) {
        return 0;  // no events removed.
    }
    // Check each bucket in this rung and cancel out events.
    int numRemoved = 0;
    for (size_t bktNum = currBucket; (bktNum < bucketList.size()); bktNum++) {
        if (!bucketList[bktNum].empty() &&
            (rStartTS + (bktNum + 1) * bucketWidth) >= sendTime) {
            // Have the bucket remove necessary event(s) and update stats
            numRemoved +=
                bucketList[bktNum].remove_after(sender, sendTime
                                                LQ2T_STATS(COMMA ceScanRung));
        }
    }
    // Update events left in this rung.
    rungEventCount -= numRemoved;
    DEBUG(validateEventCounts());
    return numRemoved;
}

int
xxxx::TwoTierRung::remove(xxxx::AgentID receiver
                          LQ2T_STATS(COMMA Avg& ceScanRung)) {
    if (empty()) {
        return 0;  // no events to be removed.
    }
    // Have each bucket in the rung remove events
    int numRemoved = 0;
    for (size_t bktNum = currBucket; (bktNum < bucketList.size()); bktNum++) {
        if (!bucketList[bktNum].empty()) {
            // This stat needs to be tracked by the bucket and not here.
            LQ2T_STATS(ceScanRung += bucketList[bktNum].size());
            // Remove appropriate set of events.
            numRemoved += bucketList[bktNum].remove(receiver);
        }
    }
    rungEventCount -= numRemoved;
    DEBUG(validateEventCounts());
    return numRemoved;
}

void
xxxx::TwoTierRung::validateEventCounts() const {
    int numEvents = 0;
    for (const auto& bucket : bucketList)  {
        numEvents += bucket.size();
    }
    if (numEvents != rungEventCount) {
        DEBUG(std::cout << "Rung event count mismatch! Expecting: "
                        << rungEventCount << " events, but found: "
                        << numEvents << "." << std::endl);
        ASSERT(numEvents == rungEventCount);
    }
}
```

```
451 // Method called just before a rung is removed from the ladder queue.
452 void
453 xxxx::TwoTierRung::updateStats(Avg& avgBktCnt) const {
454     LQ2T_STATS(avgBktCnt += maxBkts);
455 }
456
457 // This method is used only for troubleshooting/debugging purposes.
458 bool
459 xxxx::TwoTierRung::haveBefore(const Time recvTime) const {
460     for (size_t i = 0; (i < bucketList.size()); i++) {
461         if (bucketList[i].haveBefore(recvTime)) {
462             return true;
463         }
464     }
465     return false;
466 }
467
468 void
469 xxxx::TwoTierRung::prettyPrint(std::ostream& os) const {
470     // Compute minimum, maximum, empty, and average bucket sizes.
471     size_t minBkt = -1U, maxBkt = 0, emptyBkt = 0, sizeSum = 0;
472     for (const TwoTierBucket& bkt : bucketList) {
473         if (!bkt.empty()) {
474             minBkt   = std::min(minBkt, bkt.size());
475             maxBkt   = std::max(maxBkt, bkt.size());
476             sizeSum += bkt.size();
477         } else {
478             emptyBkt++;
479         }
480     }
481     double avgBktSz = sizeSum / (double) (bucketList.size() - emptyBkt);
482     os << "start time="    << rStartTS     << ", curr time=" << rCurrTS
483        << ", bkt. width=" << bucketWidth << ", bkt count=" << bucketList.size()
484        << ", curr buckt=" << currBucket  << ", events="     << rungEventCount
485        << ", min bkt="    << minBkt      << ", maxBkt="     << maxBkt
486        << ", empty bkt="  << emptyBkt    << ", avg size="   << avgBktSz
487        << std::endl;
488 }
489
490 // --------------------[ TwoTierLadderQueue methods ]--------------------
491
492 // The maximum number of rungs typically allowed in the ladder.  This
493 // value is set to 8 by default based on Tang et. al.  It can be set
494 // via command-line parameter --lq-max-rungs 8.
495 size_t xxxx::TwoTierLadderQueue::MaxRungs = 8;
496
497 void
498 xxxx::TwoTierLadderQueue::reportStats(std::ostream& os) {
499     UNUSED_PARAM(os);
500     LQ2T_STATS({
501             // Collect final bucket counts from the ladder
502             for (size_t i = 0; (i < nRung); i++) {
503                 ladder[i].updateStats(avgBktCnt);
504             }
505             // Compute net number of compares for ladderQ
506             // const long comps = log2(botLen.getMean()) * botLen.getSum();
507             // std::make_heap has 3N time complexity.
508             const long comps = 3 * botLen.getSum() +
509                 log2(botLen.getMean()) * botLen.getSum() / 3;
510             os << "Events cancelled from top  : "    << ceTop
511                << "\nEvents scanned in top      : " << ceScanTop
512                << "\nEvents cancelled from ladder: " << ceLadder
513                << "\nEvents scanned from ladder : " << ceScanLadder
514                << "\nEvents cancelled from bottom: " << ceBot
515                << "\nEvents scanned from bottom : " << ceScanBot
516                << "\nNo cancel scans of bottom  : " << ceNoCanScanBot
517                << "\nInserts into top           : " << insTop
518                << "\nInserts into rungs         : " << insLadder
519                << "\nInserts into bottom        : " << insBot
520                << "\nMax rung count             : " << maxRungs
521                << "\nAverage #buckets per rung  : " << avgBktCnt
522                << "\nAverage bottom size        : " << botLen
523                << "\nMax bottom size            : " << maxBotSize
524                << "\nAverage bucket width       : " << avgBktWidth
525                << "\nBottom to rung operations  : " << botToRung
```



```
526                << "\nCompare estimate           : " << comps
527                << std::endl;
528         });
529 }
530
531 void
532 xxxx::TwoTierLadderQueue::enqueue(xxxx::Event* event) {
533     if (top.getStartTime() < event->getReceiveTime()) {
534         DEBUG(std::cout << "Added to top: " << *event << std::endl);
535         top.add(event);
536         LQ2T_STATS(insTop++);
537         return;
538     }
539     // Try to see if the event fits in the ladder. nRung is max rung index
540     size_t rung = 0;
541     while ((rung < nRung) && !ladder[rung].canContain(event)) {
542         ASSERT((rung == 0) || ladder[rung].empty() ||
543                (ladder[rung - 1].getCurrTime() >= ladder[rung].getCurrTime()));
544         rung++;
545     }
546     if (rung < nRung) {
547         DEBUG(ASSERT(bottom.empty() ||
548                      (event->getReceiveTime() > bottom.maxTime())));
549         ladder[rung].enqueue(event);
550         ladderEventCount++;  // Track events added to the ladder
551         DEBUG(std::cout << "Added to rung " << rung  << "(max bottom: "
552                         << bottom.maxTime() << "): " << *event << "\n");
553         LQ2T_STATS(insLadder++);
554         return;
555     }
556     // Event does not fit in the ladder. It must go into bottom.
557     // However, to ensure good performance we must keep bottom short.
558     if ((bottom.size() > LQ2T_THRESH) && (bottom.getTimeRange() > 0)) {
559         // Move events from bottom into ladder rung
560         rung = createRungFromBottom();
561         ASSERT(rung == nRung - 1);
562         ASSERT(rung < ladder.size());
563         // Due to rollback-reprocessing the event may be even
564         // earlier than the last rung we just created!
565         if (ladder[rung].canContain(event)) {
566             ladder[rung].enqueue(event);
567             ladderEventCount++;
568             DEBUG(std::cout << "Added to rung " << rung  << "(max bottom: "
569                             << bottom.maxTime() << "): " << *event << "\n");
570             LQ2T_STATS(insLadder++);
571             return;
572         }
573     }
574     // At this point, event must go into bottom, so enqueue it.
575     bottom.enqueue(event);
576     LQ2T_STATS(maxBotSize = std::max(maxBotSize, bottom.size()));
577     DEBUG(ASSERT(!haveBefore(bottom.first_event()->getReceiveTime())));
578     DEBUG(std::cout << "Added to bottom: " << *event << std::endl);
579     LQ2T_STATS(insBot++);
580 }
581
582 // Implementation close to the version from the paper.
583 xxxx::TwoTierBucket&&
584 xxxx::TwoTierLadderQueue::recurseRung() {
585     ASSERT(!empty());
586     ASSERT(nRung > 0);
587     ASSERT(!ladder.empty());
588     // Now the last rung in ladder is the rung that has the next
589     // bucket of events.
590     xxxx::Time bktTime    = 0;  // set by removeNextBucket call below
591     TwoTierRung& lastRung = ladder[nRung - 1];
592     TwoTierBucket&& bkt   = lastRung.removeNextBucket(bktTime);
593     ASSERT(!bkt.empty());
594     ASSERT(!ladder.empty());
595     // Check and create new rung in the ladder if the bucket is large.
596     if ((bkt.size() > LQ2T_THRESH) && (nRung < MaxRungs)) {
597         // Note: Here bucket width can dip a bit low. But that is
598         // needed to ensure consistent ladder setup.
599         const double bucketWidth = (lastRung.getBucketWidth() + bkt.size() -
600                                     1.0) / bkt.size();
```

```
601         // Create a new rung in the ladder
602         nRung++;
603         if (nRung > ladder.size()) {
604             ladder.push_back(TwoTierRung(std::move(bkt), bktTime, bucketWidth));
605             ASSERT(nRung == ladder.size());
606         } else {
607             ladder[nRung - 1].move(std::move(bkt), bktTime, bucketWidth);
608         }
609         DEBUG(std::cout << "2. Bucket width: " << bucketWidth << std::endl);
610         LQ2T_STATS(avgBktWidth += bucketWidth);
611         LQ2T_STATS(maxRungs = std::max(maxRungs, nRung));
612         return recurseRung();  // Recurse now looking at newly added rung
613     }
614     // Track events being removed from the ladder
615     ladderEventCount -= bkt.size();
616     ASSERT(ladderEventCount >= 0);
617     // Return bucket being removed.
618     return std::move(bkt);
619 }
620
621 // Move events from ladder (or top) into bottom.
622 void
623 xxxx::TwoTierLadderQueue::populateBottom() {
624     if (!bottom.empty()) {
625         return;
626     }
627     if (ladderEventCount == 0) {   // nRung == -1
628         if (top.empty()) {
629             // There are no events in the ladder queue in this case
630             ASSERT(empty());
631             return;
632         }
633         // Move all events from top into buckets in first rung of the ladder!
634         nRung++;
635         ASSERT(nRung == 1);
636         ladderEventCount += top.size();     // Track events in ladder
637         // Move events to ladder
638         if (nRung > ladder.size()) {
639             ladder.push_back(TwoTierRung(std::move(top)));
640             ASSERT(nRung == ladder.size());
641         } else {
642             ladder[nRung - 1].move(std::move(top), top.getMinTime(),
643                                    top.getBucketWidth());
644         }
645         // Reset top counters and update the values of topStart for
646         // next Epoch
647         top.reset(top.getMaxTime());
648         LQ2T_STATS(maxRungs = std::max(maxRungs, nRung));
649         LQ2T_STATS(avgBktWidth += ladder.back().getBucketWidth());
650         DEBUG(std::cout << "3. Bucket width: "
651                         << ladder.back().getBucketWidth() << std::endl);
652         DEBUG(prettyPrint(std::cout));
653         ASSERT(top.empty());
654     }
655     // Bottom is empty. So we need to move events from the current
656     // bucket in the ladder to bottom.
657     ASSERT(!ladder.empty());
658     ASSERT(bottom.empty());
659     bottom.enqueue(recurseRung());  // Transfer bucket_k into bottom
660     ASSERT(!bottom.empty());
661     LQ2T_STATS(maxBotSize = std::max(maxBotSize, bottom.size()));
662     DEBUG(ASSERT(!haveBefore(bottom.first_event()->getReceiveTime())));
663     LQ2T_STATS(botLen += bottom.size());
664     // Clear out the rungs if we have used-up the last bucket in the ladder.
665     while (nRung > 0 && ladder[nRung - 1].empty()) {
666         LQ2T_STATS(ladder[nRung - 1].updateStats(avgBktCnt));
667         nRung--;  // Logically remove rung from ladder
668     }
669 }
670
671 int
672 xxxx::TwoTierLadderQueue::createRungFromBottom() {
673     ASSERT(!bottom.empty());
674     ASSERT(bottom.getTimeRange() > 0);
675     DEBUG(std::cout << "Moving events from bottom to a new rung. Bottom has "
```



```
676                     << bottom.size() << " events." << std::endl);
677     // Compute the start time and bucket width for the rung.  Note
678     // that with rollbacks, ladder can be empty and that situation
679     // needs to be handled.
680     const double bucketWidth = (ladder.empty() ? bottom.getBucketWidth() :
681                                 ladder[nRung - 1].getBucketWidth());
682     // The paper computes rStart as RCur[NRung-1].  However, due to
683     // rollback-reprocessing the bottom may have events that are below
684     // RCur[NRung-1].  Consequently, we use the minimum of the two
685     // values as as rstart
686     const Time ladBkTime = ((nRung > 0) ? ladder[nRung - 1].getCurrTime() :
687                             TIME_INFINITY);
688     const Time rStart = std::min(ladBkTime,
689                                  bottom.first_event()->getReceiveTime());
690     ASSERT(rStart < ladBkTime);
691     ASSERT(bottom.maxTime() < ladBkTime);
692     ASSERT(bottom.maxTime() <= top.getStartTime());
693     // Create a new rung and add it to the ladder.
694     DEBUG(std::cout << "Moving bottom to rung. Events: " << bottom.size()
695                     << ", rStart = " << rStart << ", bucketWidth = "
696                     << (bucketWidth / bottom.size()) << std::endl);
697     ladderEventCount += bottom.size();  // Update ladder event count
698     LQ2T_STATS(botToRung += bottom.size());
699     const double bktWidth = (bucketWidth + bottom.size() - 1.0) / bottom.size();
700     DEBUG(std::cout << "bktWidth = " << bktWidth << std::endl);
701     // Add rung and move move bottom into the last rung of the ladder.
702     nRung++;
703     if (nRung > ladder.size()) {
704         ladder.push_back(TwoTierRung(std::move(bottom), rStart, bktWidth));
705         ASSERT(nRung == ladder.size());
706     } else {
707         ladder[nRung - 1].move(std::move(bottom), rStart, bktWidth);
708     }
709     DEBUG(std::cout << "1. Bucket width: " << bktWidth << std::endl);
710     LQ2T_STATS(avgBktWidth += bktWidth);
711     LQ2T_STATS(maxRungs = std::max(maxRungs, nRung)); // Track max rungs
712     ASSERT(bottom.empty());
713     return nRung - 1;
714 }
715
716 int
717 xxxx::TwoTierLadderQueue::remove_after(xxxx::AgentID sender,
718                                        const Time sendTime) {
719     // Check and cancel entries in top rung.
720     int numRemoved = top.remove_after(sender, sendTime
721                                       LQ2T_STATS(COMMA ceScanTop));
722     LQ2T_STATS(ceTop += numRemoved);
723     // Cancel out events in each rung of the ladder.
724     for (size_t rung = 0; (rung < nRung); rung++) {
725         const int rungEvtRemoved =
726             ladder[rung].remove_after(sender, sendTime
727                                       LQ2T_STATS(COMMA ceScanLadder));
728         ladderEventCount  -= rungEvtRemoved;
729         numRemoved        += rungEvtRemoved;
730         LQ2T_STATS(ceLadder += rungEvtRemoved);
731     }
732     // Clear out the rungs in ladder that are now empty after event
733     // cancellations.
734     while (nRung > 0 && ladder[nRung - 1].empty()) {
735         LQ2T_STATS(ladder[nRung - 1].updateStats(avgBktCnt));
736         nRung--;  // Logically remove rung from ladder
737     }
738     // Save original size of bottom to track stats.
739     LQ2T_STATS(const size_t botSize  = bottom.size());
740     // Cancel events from bottom.
741     const int botRemoved  = bottom.remove_after(sender, sendTime);
742     if (botRemoved > -1) {
743         numRemoved += botRemoved;
744         // Update statistics counters
745         LQ2T_STATS(ceBot += botRemoved);
746         LQ2T_STATS(ceScanBot += botSize);
747         LQ2T_STATS((botRemoved == 0) ? (ceNoCanScanBot += botSize) : 0);
748     }
749     return numRemoved;
750 }
```

```
751
752 // This method is purely for debugging. So performance is not
753 // important
754 bool
755 xxxx::TwoTierLadderQueue::haveBefore(const Time recvTime,
756                                      const bool checkBottom) const {
757     // Check top
758     if (top.haveBefore(recvTime)) {
759         std::cout << "Top has event that is <= " << recvTime << std::endl;
760         prettyPrint(std::cout);
761         return true;
762     }
763     // Check each rung of the ladder
764     for (size_t rung = 0; (rung < nRung); rung++) {
765         if (ladder[rung].haveBefore(recvTime)) {
766             std::cout << "Rung #" << rung << " has event that is <= "
767                       << recvTime << std::endl;
768             prettyPrint(std::cout);
769             return true;
770         }
771     }
772     // Check bottom rung.
773     if (checkBottom && bottom.haveBefore(recvTime)) {
774         std::cout << "Bottom has event that is <= " << recvTime << std::endl;
775         prettyPrint(std::cout);
776         return true;
777     }
778     // When control drops here it mean the whole 2-tier ladder queue
779     // does not have an event with timestamp lower than recvTime.
780     return false;
781 }
782
783 // ------------------------[  EventQueue implementation ]----------------------
784
785 void*
786 xxxx::TwoTierLadderQueue::addAgent(xxxx::Agent* agent) {
787     UNUSED_PARAM(agent);
788     return NULL;  // 2-tier queue has no cross-references to store in agent
789 }
790
791 void
792 xxxx::TwoTierLadderQueue::removeAgent(xxxx::Agent* agent) {
793     ASSERT( agent != NULL );
794     const AgentID receiver = agent->getAgentID();
795     // Remove events for agent from top
796     LQ2T_STATS(ceScanTop += top.size());
797     int numRemoved    = top.remove(receiver);
798     LQ2T_STATS(ceTop += numRemoved);
799
800     // Next remove events for agent from all the rungs in the ladder
801     for (TwoTierRung& rung : ladder) {
802         int rungEvtRemoved = rung.remove(agent->getAgentID()
803                                          LQ2T_STATS(COMMA ceBot));
804         ladderEventCount  -= rungEvtRemoved;
805         numRemoved        += rungEvtRemoved;
806         LQ2T_STATS(ceLadder += rungEvtRemoved);
807     }
808     // Finally remove events from bottom for the agent.
809     LQ2T_STATS(const size_t botSize = bottom.size());
810     const int botRemoved  = bottom.remove(receiver);
811     LQ2T_STATS(ceScanBot += botSize);
812     LQ2T_STATS((botRemoved == 0) ? (ceNoCanScanBot += botSize) : 0);
813     numRemoved           += botRemoved;
814     LQ2T_STATS(ceBot     += botRemoved);
815 }
816
817
818 xxxx::Event*
819 xxxx::TwoTierLadderQueue::front() {
820     if (empty()) {
821         // Nothing to return.
822         return NULL;
823     }
824     if (bottom.empty()) {
825         populateBottom();
```



```
826         DEBUG(prettyPrint(std::cout));
827     }
828     ASSERT(!bottom.empty());
829     return bottom.first_event();
830 }
831
832 void
833 xxxx::TwoTierLadderQueue::dequeueNextAgentEvents(xxxx::EventContainer& events) {
834     if (empty()) {
835         // No events to dequeue.
836         return;
837     }
838     // We only dequeue from bottom. So ensure it has events in it.
839     if (bottom.empty()) {
840         // Move events from top or a ladder rung into bottom.
841         populateBottom();
842     }
843     ASSERT(!bottom.empty());
844     bottom.dequeueNextAgentEvents(events);
845     ASSERT(!events.empty());
846     DEBUG(ASSERT(!haveBefore(events.front()->getReceiveTime())));
847 }
848
849 // The main interface method used by XXXX to schedule event.
850 void
851 xxxx::TwoTierLadderQueue::enqueue(xxxx::Agent* agent, xxxx::Event* event) {
852     UNUSED_PARAM(agent);
853     event->increaseReference();
854     enqueue(event);
855 }
856
857 // Method for block addition (typically used during rollback recovery)
858 void
859 xxxx::TwoTierLadderQueue::enqueue(xxxx::Agent* agent,
860                                   xxxx::EventContainer& events) {
861     UNUSED_PARAM(agent);
862     for (auto& curr : events) {
863         enqueue(curr);
864     }
865     events.clear();
866 }
867
868 // Method to cancel all events in the 2-tier heap.
869 int
870 xxxx::TwoTierLadderQueue::eraseAfter(xxxx::Agent* dest,
871                                      const xxxx::AgentID sender,
872                                      const xxxx::Time sentTime) {
873     UNUSED_PARAM(dest);
874     return remove_after(sender, sentTime);
875 }
876
877 void
878 xxxx::TwoTierLadderQueue::prettyPrint(std::ostream& os) const {
879     // Print information on top.
880     os << "Top: Events=" << top.size()
881        << ", startTime=" << top.getStartTime()
882        << ", minTime="   << top.getMinTime()
883        << ", maxTime="   << top.getMaxTime()  << std::endl;
884     // Print info on each rung of the ladder
885     std::cout << "Ladder (rungs=" << nRung << ", size="
886               << ladder.size() << "):\n";
887     for (size_t i = 0; (i < nRung); i++) {
888         os << "[" << i << "]: ";
889         ladder[i].prettyPrint(os);
890     }
891     // Print info on bottom
892     os << "Bottom: Events=" << bottom.size()
893        << ", min="          << (!bottom.empty() ? bottom.findMinTime() : -1.0)
894        << ", max="          << (!bottom.empty() ? bottom.maxTime()     : -1.0)
895        << std::endl;
896 }
897
898 #endif
```