Thread Service

Overview

The COUGAAR Thread Services are designed to replace the direct use of Java Threads with a threading system that can be more tightly controlled. The central new interface is Schedulable, which can be thought of as taking the place of a Java native thread. Schedulables are created and returned by the ThreadService and started by client code. Unlike native Java threads, Schedulables don't necessarily run immediately. The Threading Services allow only a certain number of threads to be running at once. If a Schedulable is started when all threads slots are in use, it will be queued and will only run when enough running threads have stopped to allow it reach the head of the queue. The maximum number of running Schedulables as well as the queue ordering can be controlled by the ThreadControlService. The Thread Services can also be used to schedule Java TimerTasks.

In addition to the running of threads and tasks, the COUGAAR Thread Services offers two other features: an event-like mechanism for receiving callbacks when 'interesting' events occur (ThreadListenerService), and a certain amount of explicit control over scheduling. These interfaces, as well as the Schedulable and ThreadService interfaces, are described in detail below.

The COUGAAR Thread Services are hierarchical. Each Agent has a set of Thread Services of its own, as does the MTS and certain other Node-level components. These local Thread Services handle threading at their own level and are in turn controlled by a root Thread Service at the Node level. The inter-level control mechanism takes the form of "rights" given by the higher to the lower level services, and returned after use by the lower to the higher level services. When the higher level service has multiple children, a RightsSelector is used to choose the child service that will be the next to receive rights (round-robin by default). In principle this hierarchy could be extended to a further depth but so far we haven't found any good reason to do that in practice.

Within a level, scheduling is further subdivided into lanes. There are currently four hardwired lanes. A lane is associated with a Schedulable at creation time and remains fixed for the lifetime of the Schedulable.

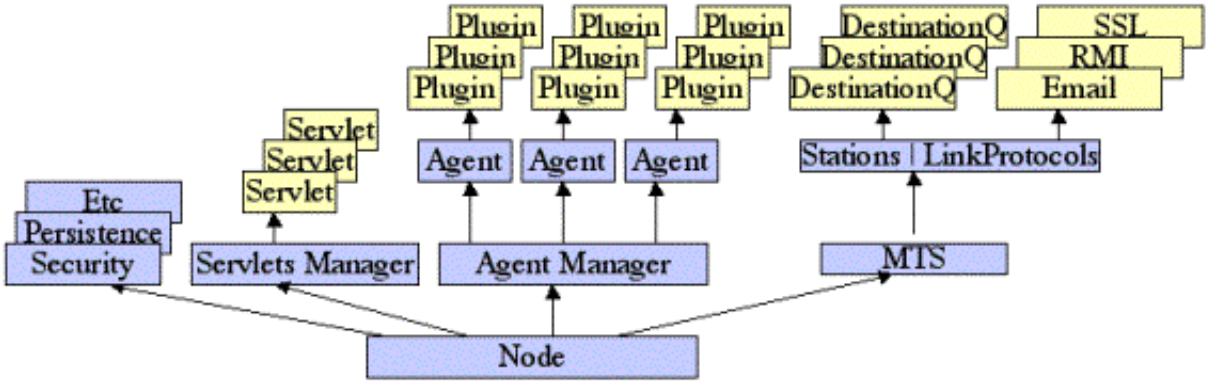
The design of the thread services has important implications for the runtime pattern of the Runnables it will be running. In particular, it's not generally a good idea for these Runnables to block. The common Java pattern of a loop with a wait or sleep call should usually be unwrapped into a simple 'strip' of code that reschedules itself if it needs to run again. Examples can be seen in Use Cases and Examples. Code which needs to block or to use an unusual amount of resources should run in the appropriate lane if at all possible.

Design and Architecture

The overall structure of the thread service system is a tree that tends to mimic the tree of Containers in COUGAAR.

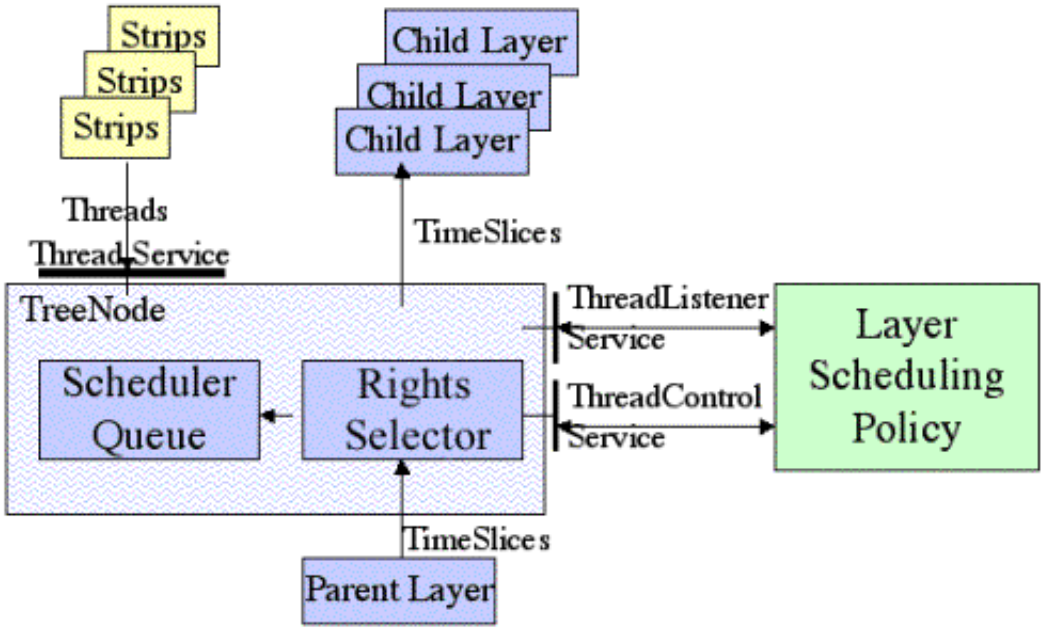
Each level of this tree includes its own trio of thread services which are directly responsible for the Threads used at that level of the hierarchy and indirectly responsible for its children.

Each level other than the root requests run-rights from its parent and only runs as many java Threads as it has rights. With the default scheduler, all requests for run-rights ultimately propagate to the root service, which keeps a count of running threads and refuses to allocate further rights if that count hits a given maximum. As each Thread finishes, its run-right is released, again propagating to the root service. The released right will then be made available to the children, using a layer-specific algorithm. Each layer can (a) consume the right itself (if the given layer has queued threads); (b) recursively give it to a child; or (c) decline to accept it.



The direct control of local threads at any given level is handled by a per-lane Scheduler. If a thread at that level wants to run, the Scheduler for its lane is asked for a right. If a right is available a true java Thread will be run; if not, the request will be queued until sufficient rights are available. The order in which items are removed from the queue depends on a Comparator, which by default uses time (i.e., fifo) but which at any time can be replaced by an arbitrarily complex and dynamic Comparator via the ThreadControlService. Schedulers use a RightsSelector to determine the possible re-allocation of a released run-right. The default RightsSelector uses a round-robin algorithm, which provides fair scheduling between the layer's own queued requests and its children. The RightsSelector can be replaced at any time via the ThreadControlService.

The hierarchy skeleton is represented by a set of TreeNodes. The TreeNode at any given level holds pointers to the level's scheduler, selector, parent node and child nodes.



Any Container whose components wish to use the thread services should provide those services locally, using ThreadServiceProvider. This will ensure that the thread service hierarchy maps properly into the Component hierarchy.

Core Internal Classes

ThreadPool

This is an implementation of a classic thread-pool. The threads it provides are defined by an inner class, PooledThread.

SchedulableObject

This is the implementation of Schedulable. It communicates with a Scheduler when it wants to request or return rights and with the ThreadPoolwhen it needs a Thread.

Scheduler

This is the simplest scheduler class. It deals with run-rights locally, completely ignoring the hierarchy. It's also the implementation of the ThreadControlService.

PropagatingScheduler

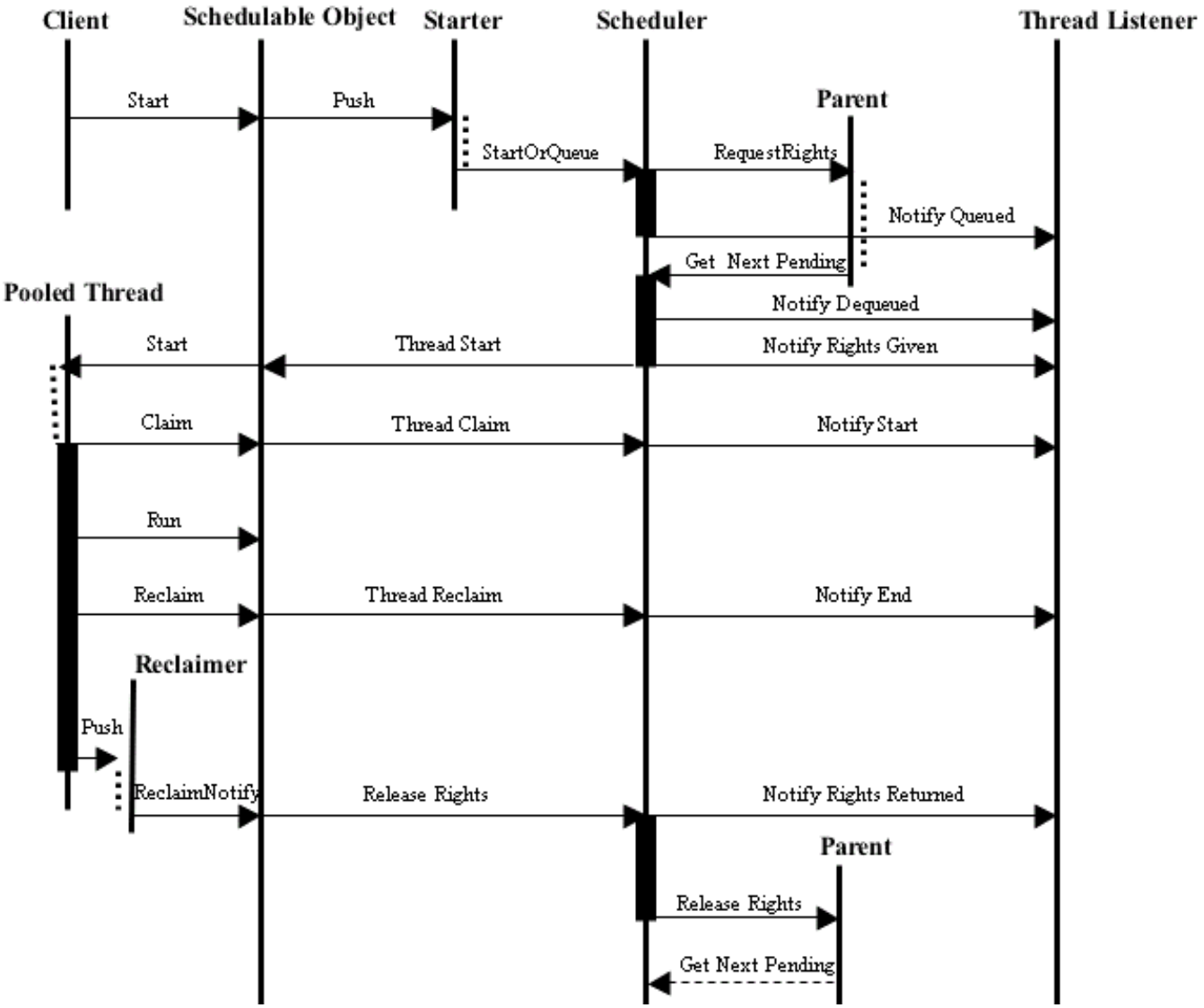
This is an extension of Scheduler that provides the standard hierarchical behavior, as described above.

ThreadListenerProxy

This is the implementation of ThreadListenerService.

ThreadServiceProxy

This is the implementation of ThreadService. It creates the Thread TreeNodeand the thread pool and links the pool to a Scheduler.



— Thread Lifecycle Execution Graph

Use Cases and Examples

Code strips: avoiding calls to wait()

As explained in the overview, Schedulables run more effectively as strips of code than as loops with a wait, since the blocking behavior of the wait will tie up a limited resource needlessly.

As an example, consider a thread-based queue pattern, which might look something the following.

```
class QueueRunner implements Runnable {
    private SimpleQueue queue;
    private Thread thread;

    QueueRunner() {
        queue = new SimpleQueue(); // make the internal queue
        thread = new Thread(this, "My Queue");
        thread.start();  // start the thread
    }

    public void run() {
        Object next = null;
        while (true) {
            while (true) {
                synchronized (queue) {
                    if (!queue.isEmpty()) {
                        next = queue.pop();
                        break; // process the next item
                    } // queue is empty, wait to be notified of new items
                    try {
                        queue.wait();
                    } catch (InterruptedException ex) {}
                }
            }
            processNext(next);
        }
    }

    public void add(Object x) {
        synchronized (queue) {
            queue.add(x);
            queue.notify(); // wake up the wait()
        }
    }
}
```

This thread will spend most of its time in the wait call, which is not at all an effective use of a limited resource. As a Schedulable it would be better written as follows:

```
class QueueRunner implements Runnable {
    private SimpleQueue queue;
    private Schedulable schedulable;

    QueueRunner() {
        queue = new SimpleQueue();    // Create the Schedulable but don't start it
        schedulable = threadService.getThread(this, this, "MyQueue");
    }

    public void run() {    // Handle all items currently queued but never block
        Object next = null;
        while (true) {
            synchronized (queue) {
                if (queue.isEmpty()) break; // done for now
                next = queue.pop();
            }
            processNext(next);
        }
    }

    void add(Object next) {
        synchronized (queue) {
            queue.add(next);
        }    // Restart the schedulable if it's not currently running.
        schedulable.start();
    }
}
```

Code strips: avoiding calls to sleep()

Another popular Java Thread pattern uses a sleep in a never-ending loop. As above, this uses up a thread resource even though the thread is rarely running.

```
class SleepingPoller implements Runnable  {
    private int period;
    private String name;

    SleepingPoller(String name, Object client, int period) {
        this.name = name;
        this.period = period;
        ThreadService ts = (ThreadService)sb.getService(this, ThreadService.class,
        ts.getThread(client, this, name).start();
        sb.releaseService(this, ThreadService.class, ts);
    }

    void initialize() {
        // Initialization code here
    }

    void executeBody() throws Exception {
        // Thread body here.
    }

    public void run() {
        initialize();
        while (true) {
            try {
                executeBody();
            } catch (Exception ex) {
                log.error("Error in thread " + name, ex);
            }

            try {
                Thread.sleep(period);
            } catch {
                InterruptedExecption (ex);
            }
        }
    }
}
```

In this case the use of sleep can be replaced by having the run() method restart the thread after a delay.

```
class Poller implements Runnable  {
     private Schedulable schedulable;
     private boolean initialized = false;
     private int period;
     private String name;
     private ThreadService ts;

     Poller(String name, Object client, int period) {
         this.name = name;
         this.period = period;
         this.ts = (ThreadService) sb.getService(this, ThreadService.class, null);
         this.schedulable = ts.getThread(client, this, name);
         schedulable.start();
     }

     void ensureInitiolized() {
         synchronized (this) {
             if (initialized) return;
             initialized = true;
         }

         initialize();
     }

     void initialize() {
         // Initialization code here
     }

     void executeBody() throws Exception {
         // Thread body here.
     }

     public void run() {
         ensureInitiolized();
         try {
             executeBody();
         } catch (Exception ex) {
             log.error("Error in thread " + name, ex);
         }

         // Restart after period ms
         schedulable.schedule(period);
     }
 }
```

Code strips: avoiding TimerTasks

TimerTasks are not controllable and should generally be avoided. Instead use the schedule methods on Schedulable for equivalent functionality. In this case the body of the task will run as a COUGAAR thread. Compare runTask() and runThreadPeriodically() below.

```
class MyPeriodicCode {
    private int period;
    private Schedulable schedulable;

    public void body() {
        // The body of code to be run periodically goes here.
    }

    // In this version body() runs in the Thread Service's Timer
    // thread, which can be problematic if it takes too long.
    void runTask() {
        ThreadService ts = sb.getService(this, ThreadService.class, null);
        TimerTask task = new TimerTask() {
            public void run() {
                body();
            }
        };
        ts.schedule(task, 0, period);
        sb.releaseService(this, ThreadService.class, ts);
    }

    // In this version body() runs periodically as a cougaar thread.
    void runThreadPeriodically() {
        ThreadService ts = sb.getService(this, ThreadService.class, null);
        Runnable code = new Runnable () {
            public void run() {
                 body();
            }
        };
        schedulable = ts.getThread(this, code, "MyPeriodicCode");

        // Restart the Schedulable every period ms
        schedulable.schedule(0, period);
        sb.releaseService(this, ThreadService.class, ts);
    }
}
```

APIs

org.cougaar.core.thread.Schedulable

- THREAD_RUNNING
- THREAD_PENDING (ie queued)

- THREAD_DISQUALIFIED (see ThreadControlService)
- THREAD_DORMANT (ie none of the above)
- THREAD_SUSPENDED (not currently used)

org.cougaar.core.service.ThreadControlService

void setDefaultLane(int lane)

Sets the default lane for any newly created Schedulables. This has no effect on extant Schedulables.

void setMaxRunningThreadCount(int count, int lane)

Sets the maximum number of Schedulables that are allowed to run at any one time (across all levels) for the given lane.

void setQueueComparator(Comparator comparator, int lane)

Sets the Comparator used by the queue at this level for the given lane to order its elements (Schedulables). The ‘smallest’ value, as determined by the Comparator, is the first element of the queue. By default the queue is sorted by time (fifo).

void setRightsSelector(RightsSelector selector, int lane)

By default, the “right” to run is handled in a round-robin fashion. This can be overridden by providing a different RightsSelector. The rights-selection mechanism is experimental and lightly tested, and shouldn’t be used (yet) except for experimenting.

boolean setQualifier(UnaryPredicate predicate, int lane)

Sets the qualifier at this level for the given lane. The qualifier is used to disqualify queued Schedulables temorarily. Any Schedulable on the queue which doesn’t satisfy the predicate is removed and held in another list. Such a Schedulable will only be returned to the queue (and thus given the opportunity to run eventually) if the qualifier is unset (ie, set to null). NB: The qualifier can only be set to a non-null value if it’s current null.

boolean setChildQualifier(UnaryPredicate predicate, int lane)

Sets the qualifier for child schedulers at this level for the given lane. The qualifier is used to prevent children from gaining rights they might otherwise have access to. This is useful to keep one child from using up all its parents rights (for example).NB: The qualifier can only be set to a non-null value if it’s current null .

void setMaxRunningThreadCount(int count)

Sets the maximum number of Schedulables that are allowed to run at any one time (across all levels) for the default lane.

void setQueueComparator(Comparator comparator)

Sets the Comparator used by the queue at this level for the default lane to order its elements (Schedulables). The ‘smallest’ value, as determined by the Comparator, is the first element of the queue. By default the queue is sorted by time (fifo).

void setRightsSelector(RightsSelector selector)

By default, the “right” to run is handled in a round-robin fashion. This can be overridden by providing a different RightsSelector. The rights-selection mechanism is experimental and lightly tested, and shouldn’t be used (yet) except for experimenting.

boolean setQualifier(UnaryPredicate predicate)

Sets the qualifier at this level for the default lane. The qualifier is used to disqualify queued Schedulables temorarily. Any Schedulable on the queue which doesn’t satisfy the predicate is removed and held in another list. Such a Schedulable will only be returned to the queue (and thus given the opportunity to run eventually) if the qualifier is unset (ie, set to null).NB: The qualifier can only be set to a non-null value if it’s current null .

boolean setChildQualifier(UnaryPredicate predicate)

Sets the qualifier for child schedulers at this level for the default lane. The qualifier is used to prevent children from gaining rights they might otherwise have access to. This is useful to keep one child from using up all its parents rights (for example).NB: The qualifier can only be set to a non-null value if it’s current null .

int getDefaultLane()

Returns the current default lane.

int runningThreadCount(int lane)

Returns the number of Schedulables that are currently running at this level for the given lane.

int pendingThreadCount(int lane)

Returns the number of Schedulables that are currently queued at this level for the given lane.

int activeThreadCount(int lane)

Returns the number of Schedulables that are currently either running or queued at this level for the given lane.

int maxRunningThreadCount(int lane)

Returns the maximum number of Schedulables that are allowed to run at any one time (across all levels) for the given lane.

int runningThreadCount()

Returns the number of Schedulables that are currently running at this level for the default lane.

int pendingThreadCount()

Returns the number of Schedulables that are currently queued at this level for the default lane.

int activeThreadCount()

Returns the number of Schedulables that are currently either running or queued at this level for the default

lane.

int maxRunningThreadCount()

Returns the maximum number of Schedulables that are allowed to run at any one time (across all levels) for the default lane.

org.cougaar.core.thread.ThreadListener

void threadQueued(Schedulable schedulable, Object consumer)

Subscribed ThreadListenersat any given level will receive this callback when any Schedulable is placed on the queue of pending 'threads' for that level. The consumer on whose behalf the Schedulable was created is also provided.

void threadDequeued(Schedulable schedulable, Object consumer)

Subscribed ThreadListenersat any given level will receive this callback when any Schedulable is popped of the queue of pending 'threads' for that level, in preparation for being run. The consumer on whose behalf the Schedulable was created is also provided.

void threadStarted(Schedulable schedulable, Object consumer

Subscribed ThreadListenersat any given level will receive this callback when any Schedulable at that level starts running. The consumer on whose behalf the Schedulable was created is also provided.

void threadStopped(Schedulable schedulable, Object consumer)

Subscribed ThreadListenersat any given level will receive this callback when any Schedulable at that level stops running. The consumer on whose behalf the Schedulable was created is also provided.

void rightGiven(String consumer);

Subscribed ThreadListenersat any given level will receive this callback when the next higher Thread Services have granted this level a "right" for the given thread consumer. No Schedulable has claimed the right at this point so none is passed in the callback.

void rightReturned(String consumer)

Subscribed ThreadListeners at any given level will receive this callback when that level returns a "right" to the next higher level. The Schedulable which used the right is already back in the thread pool at this point, so it isn't passed in the callback.

org.cougaar.core.service.ThreadListenerService

void addListener(ThreadListener listener, int lane)

Adds the given listener to the list of those which will receive callback notifications for 'interesting' thread events (queued, dequeued, started, stopped, etc) at this level of the Thread Services, for the given lane.

void removeListener(ThreadListener listener, int lane)

Removes the given listener from the list of those which will receive callback notifications for 'interesting' thread events (queued, dequeued, started, stopped, etc) at this level of the Thread Services, for the given

lane.

void addListener(ThreadListener listener)

Adds the given listener to the list of those which will receive callback notifications for ‘interesting’ thread events (queued, dequeued, started, stopped, etc) at this level of the Thread Services, for the default lane.

void removeListener(ThreadListener listener)

Removes the given listener from the list of those which will receive callback notifications for ‘interesting’ thread events (queued, dequeued, started, stopped, etc) at this level of the Thread Services, for the default lane.

org.cougaar.core.service.ThreadService

Here is an example usage:

```
try{
   SchedulableStatus.beginNetIO("RMI reference decode");
   object = RMIRemoteObjectDecoder.decode(ref);
} catch (Throwable ex) {
   loggingService.error("Can't decode URI " +ref, ex);
} finally {
   SchedulableStatus.endBlocking();
}
```

org.cougaar.core.thread.SchedulableStatus

The ScheduleStatus class is used to dynamically mark when a thread ventures into a blocking region. It is preferable to not allow threads to block and to refactor your component into a non-blocking form. Sometimes it is not possible to refactor out the blocking call. In these cases, the region of the code that blocks should be marked when the pooled thread enters and leaves the blocking region.

To help debugging, the blocking regions are displayed as part are as part of the TopServlet and the RogueThreadDetector. Since the Schedulable is marked with the blocking region, Cougaar thread scheduler policies can also use the status to help determine which Schedulable to run next or the max number of pooled threads.

The following status types are defined: OTHER, WAIT, FILEIO, NETIO
A NOT_BLOCKING type is a negative number.

void beginBlocking(int type, String excuse)

Mark the current running Schedulable as Blocking, with a given type and excuse. The static method finds the Schedulable for the running thread, so you do not need to do any bookkeeping your self. This call is cheap and should be used as close to the blocking region as possible

void beginWait(String excuse)

Sets status type to WAIT

void beginFileIO(String excuse)

Sets status type to FILEIO

void beginNetIO(String excuse)

Sets status type to NETIO

void endBlocking()

Call this method when leaving the blocking region

Here is an example usage:

```
try {
    SchedulableStatus.beginNetIO("RMI reference decode");
    object = RMIRemoteObjectDecoder.decode(ref);
} catch (Throwable ex) {
    loggingService.error("Can't decode URI " +ref, ex);
} finally {
    SchedulableStatus.endBlocking();
}
```

Configuration

Command-line (-D) Properties

org.cougaar.thread.trivial

Set this to “true” to use the extremely simple Thread Service implementation (with no Schedulers, control, or listener callbacks). Later this will be handled via Component Descriptions.

Component Parameters

BestEffortAbsCapacity

Set this to an integer to specify the maximum number of threads that can ever run in the BEST_EFFORT_LANE. This only has relevance for the Node-level Thread Service. The default value is 300.

WillBlockAbsCapacity

Set this to an integer to specify the maximum number of threads that can ever run in the WILL_BLOCK_LANE. This only has relevance for the Node-level Thread Service. The default value is 30.

CpuIntenseAbsCapacity

Set this to an integer to specify the maximum number of threads that can ever run in the CPU_INTENSE_LANE. This only has relevance for the Node-level Thread Service. The default value is 2.

WellBehavedAbsCapacity

Set this to an integer to specify the maximum number of threads that can ever run in the

WELL_BEHAVED_LANE. This only has relevance for the Node-level Thread Service. The default value is 2.

Servlets

org.cougaar.core.thread.TopPlugin

Dumps the currently running or queued Schedulables. It does not depend on the Metrics Service. This should be loaded into the NodeAgent.

org.cougaar.core.thread.AgentSensorPlugin

Publishes per-agent load average data into the Metrics Service. This should be loaded into the NodeAgent.

org.cougaar.core.thread.ThreadWellBehavedPlugin

Loading this plugin into an Agent causes that Agent to use the WELL_BEHAVED_LANE as its default. The plugin also accepts a parameter “defaultLane” to set some other initial default. For now the parameter value should be given directly as an integer, not symbolically.

Cougaar

Cougaar Agent Architecture Open-Source site

Configuration

Thread Service – Configuration

Command-line (-D) Properties

org.cougaar.thread.trivial

Set this to “true” to use the extremely simple Thread Service implementation (with no Schedulers, control, or listener callbacks). Later this will be handled via Component Descriptions.

Component Parameters

BestEffortAbsCapacity

Set this to an integer to specify the maximum number of threads that can ever run in the `BEST_EFFORT_LANE`. This only has relevance for the Node-level Thread Service. The default value is 300.

WillBlockAbsCapacity

Set this to an integer to specify the maximum number of threads that can ever run in the `WILL_BLOCK_LANE`. This only has relevance for the Node-level Thread Service. The default value is 30.

CpuIntenseAbsCapacity

Set this to an integer to specify the maximum number of threads that can ever run in the `CPU_INTENSE_LANE`. This only has relevance for the Node-level Thread Service. The default value is 2.

WellBehavedAbsCapacity

Set this to an integer to specify the maximum number of threads that can ever run in the `WELL_BEHAVED_LANE`. This only has relevance for the Node-level Thread Service. The default value is 2.

Servlets

org.cougaar.core.thread.TopPlugin

Dumps the currently running or queued `Schedulables`. It does not depend on the Metrics Service. This should be loaded into the NodeAgent.

org.cougaar.core.thread.AgentSensorPlugin

Publishes per-agent load average data into the Metrics Service. This should be loaded into the NodeAgent.

org.cougaar.core.thread.ThreadWellBehavedPlugin

Loading this plugin into an Agent causes that Agent to use the WELL_BEHAVED_LANE as its default. The plugin also accepts a parameter “defaultLane” to set some other initial default. For now the parameter value should be given directly as an integer, not symbolically.

Cougaar

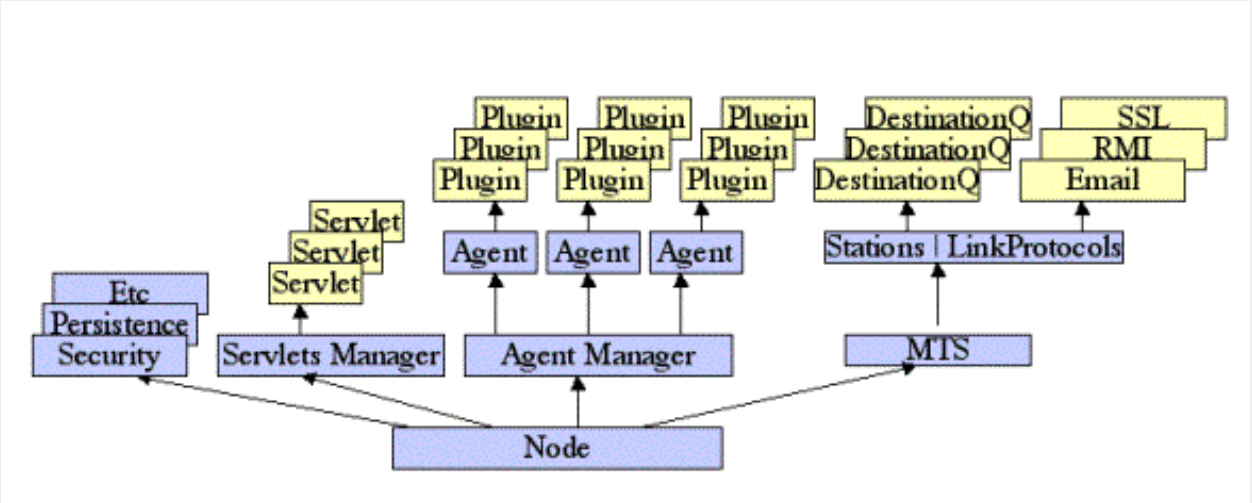
Cougaar Agent Architecture Open-Source site

Design and Architecture

Design and Architecture

The overall structure of the thread service system is a tree that tends to mimic the tree of Containers in COUGAAR. Each level of this tree includes its own trio of thread services which are directly responsible for the Threads used at that level of the hierarchy and indirectly responsible for its children.

Each level other than the root requests run-rights from its parent and only runs as many java Threads as it has rights. With the default scheduler, all requests for run-rights ultimately propogate to the root service, which keeps a count of running threads and refuses to allocate further rights if that count hits a given maximum. As each Thread finishes, its run-right is released, again propagating to the root service. The released right will then be made available to the children, using a layer-specific algorithm. Each layer can (a) consume the right itself (if the given layer has queued threads); (b) recursively give it to a child; or (c) decline to accept it.

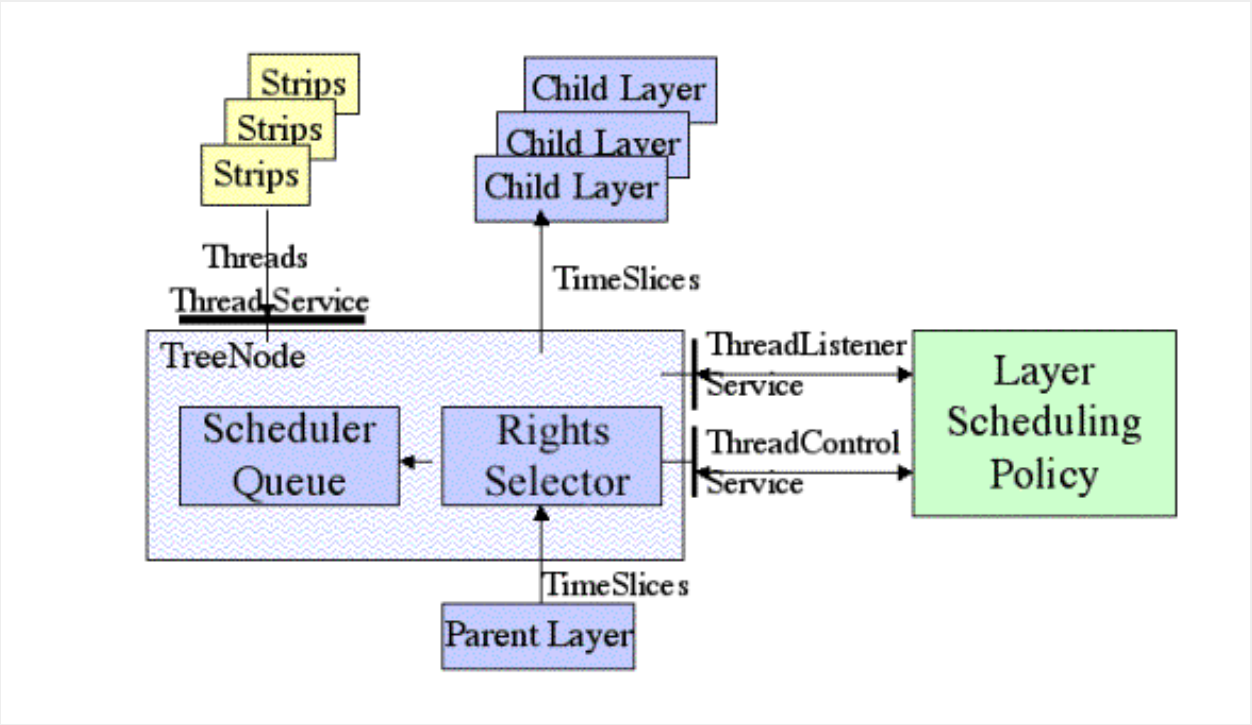


— Thread Service Design

The direct control of local threads at any given level is handled by a per-lane Scheduler. If a thread at that level wants to run, the Scheduler for its lane is asked for a right. If a right is available a true java Thread will be run; if not, the request will be queued until sufficient rights are available. The order in which items are removed from the queue depends on a Comparator, which by default uses time (i.e., fifo) but which at any time can be replaced by an arbitrarily complex and dynamic Comparator via the ThreadControlService. Schedulers use a RightsSelector to determine the possible re-allocation of a released run-right. The default RightsSelector uses a round-robin algorithm, which provides fair scheduling between the layer's own queued requests and its children. The RightsSelector can be replaced at any time

via the ThreadControlService.

The hierarchy skeleton is represented by a set of TreeNodes. The TreeNode at any given level holds pointers to the level's scheduler, selector, parent node and child nodes.



— Thread Services Layer Architecture

Any Container whose components wish to use the thread services should provide those services locally, using ThreadServiceProvider. This will ensure that the thread service hierarchy maps properly into the Component hierarchy.

Core Internal Classes

ThreadPool

This is an implementation of a classic thread-pool. The threads it provides are defined by an inner class, PooledThread.

SchedulableObject

This is the implementation of Schedulable. It communicates with a Scheduler when it wants to request or return rights and with the ThreadPool when it needs a Thread.

Scheduler

This is the simplest scheduler class. It deals with run-rights locally, completely ignoring the hierarchy. It's also the implementation of the ThreadControlService.

PropagatingScheduler

This is an extension of Scheduler that provides the standard hierarchical behavior, as described above.

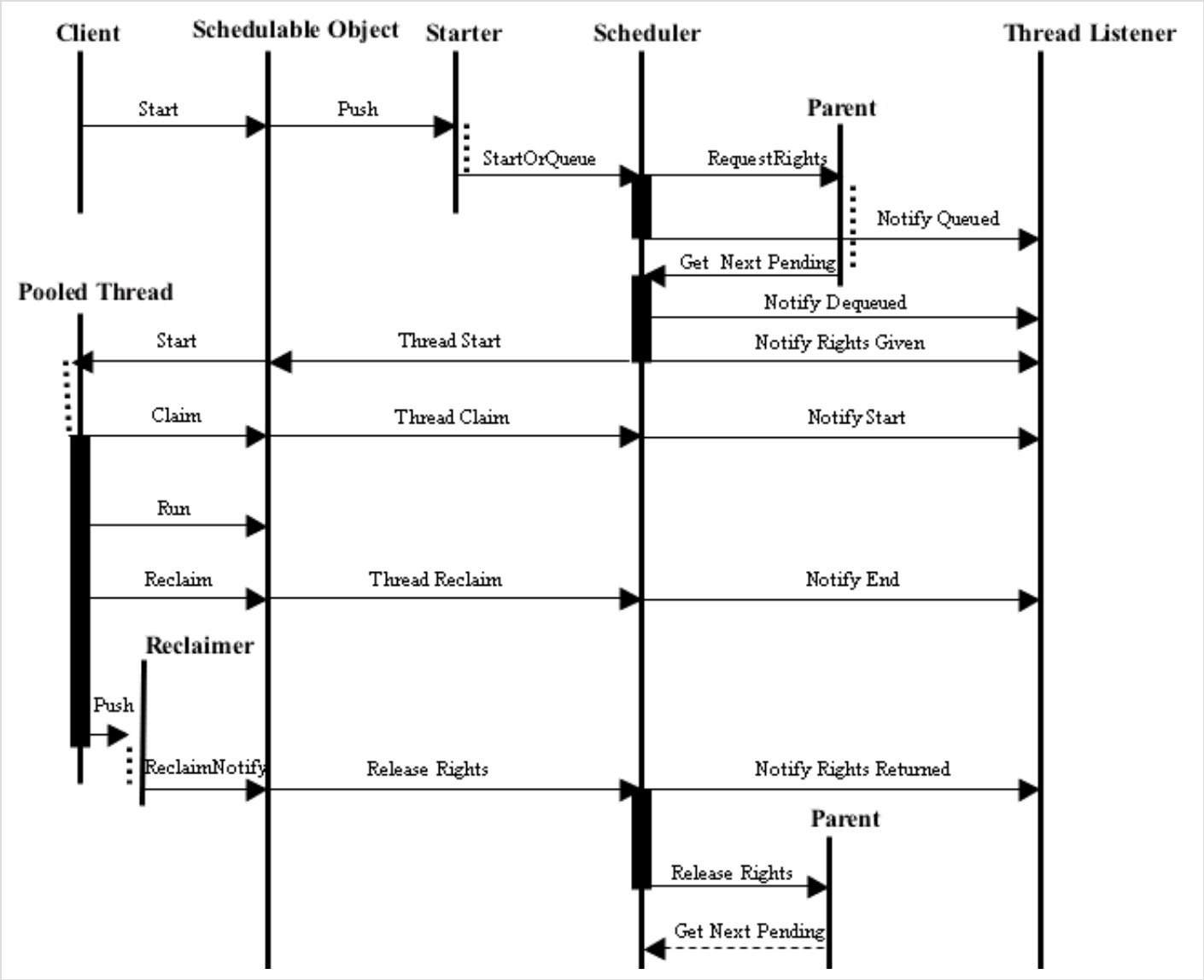
ThreadListenerProxy

This is the implementation of ThreadListenerService.

ThreadServiceProxy

This is the implementation of ThreadService. It creates the Thread TreeNode and the thread pool and links the pool to a Scheduler.

Interaction between Internal Classes



Cougaar

Cougaar Agent Architecture Open-Source site

Schedulable

org.cougaar.core.thread.Schedulable

`void start()`

Starting a `Schedulable` is conceptually the same as starting a Java native thread, but the behavior is slightly different in two ways. First, if no thread resources are available, the `Schedulable` will be queued instead of running righy away. It will only run when enough resources have become available for it to reach the head of the queue. Second, if the `Schedulable` is running at the time of the call, this call will cause the `Schedulable` restart itself after the current run finishes (unless it's cancelled in the meantime). If the Schedulable is pending at the time of the call, this method is a no-op.

`void schedule(long delay)`

Equivalent to calling `start()`after the given delay (in millis).

`void schedule(long delay, long period)`

Runs a `Schedulable` periodically at the period given, where each run is equivalent to `start()`(ie it may not start immediately). The delay applies to the first run.

`void scheduleAtFixedRate(long delay, long period)`

This is to the method above as the method by the same name on `TimerTask` is to the `schedule` method on that class (see Sun's TimerTask javadoc).

`void cancelTimer()`

This method cancels the active periodic schedule. It does not cancel the `Schedulable` entirely (see `cancel()`).

`boolean cancel()`

Cancelling a `Schedulable` will prevent it starting if it's currently queued or from restarting if it was scheduled to do so. It will not cancel the current run. This method also runs `cancelTimer`.

`int getState()`

Returns the current state of the `Schedulable`. The states are defined in org.cougaar.core.thread.CougaarThread and consist of the following:

- THREAD_RUNNING
- THREAD_PENDING (ie queued)

- THREAD_DISQUALIFIED (see thread disqualification)
- THREAD_DORMANT (ie none of the above)
- THREAD_SUSPENDED (not currently used)

`Object getConsumer()`

Returns the COUGAAR entity for which the `ThreadService` made this `Schedulable`. See thread creation.

`int getLane()`

Returns the lane that this `Schedulable` was assigned to. See thread creation. Lane constants can be found in the ThreadService interface.

Cougaar

Cougaar Agent Architecture Open-Source site

Schedulable

org.cougaar.core.thread.Schedulable

`void start()`

Starting a `Schedulable` is conceptually the same as starting a Java native thread, but the behavior is slightly different in two ways. First, if no thread resources are available, the `Schedulable` will be queued instead of running righy away. It will only run when enough resources have become available for it to reach the head of the queue. Second, if the `Schedulable` is running at the time of the call, this call will cause the `Schedulable` restart itself after the current run finishes (unless it's cancelled in the meantime). If the Schedulable is pending at the time of the call, this method is a no-op.

`void schedule(long delay)`

Equivalent to calling `start()`after the given delay (in millis).

`void schedule(long delay, long period)`

Runs a `Schedulable` periodically at the period given, where each run is equivalent to `start()`(ie it may not start immediately). The delay applies to the first run.

`void scheduleAtFixedRate(long delay, long period)`

This is to the method above as the method by the same name on `TimerTask` is to the `schedule` method on that class (see Sun's TimerTask javadoc).

`void cancelTimer()`

This method cancels the active periodic schedule. It does not cancel the `Schedulable` entirely (see `cancel()`).

`boolean cancel()`

Cancelling a `Schedulable` will prevent it starting if it's currently queued or from restarting if it was scheduled to do so. It will not cancel the current run. This method also runs `cancelTimer`.

`int getState()`

Returns the current state of the `Schedulable`. The states are defined in org.cougaar.core.thread.CougaarThread and consist of the following:

- THREAD_RUNNING
- THREAD_PENDING (ie queued)

- THREAD_DISQUALIFIED (see thread disqualification)
- THREAD_DORMANT (ie none of the above)
- THREAD_SUSPENDED (not currently used)

`Object getConsumer()`

Returns the COUGAAR entity for which the `ThreadService` made this `Schedulable`. See thread creation.

`int getLane()`

Returns the lane that this `Schedulable` was assigned to. See thread creation. Lane constants can be found in the ThreadService interface.

Cougaar

Cougaar Agent Architecture Open-Source site

SchedulableStatus

org.cougaar.core.thread.SchedulableStatus

The ScheduleStatus class is used to dynamically mark when a thread ventures into a blocking region. It is preferable to not allow threads to block and to refactor your component into a non-blocking form. Sometimes it is not possible to refactor out the blocking call. In these cases, the region of the code that blocks should be marked when the pooled thread enters and leaves the blocking region.

To help debugging, the blocking regions are displayed as part are as part of the TopServlet and the RogueThreadDetector. Since the `Schedulable` is marked with the blocking region, Cougaar thread scheduler policies can also use the status to help determine which `Schedulable` to run next or the max number of pooled threads.

The following status types are defined: `OTHER, WAIT, FILEIO, NETIO`
A `NOT_BLOCKING` type is a negitive number.

`void beginBlocking(int type, String excuse)`
: Mark the current running `Schedulable` as Blocking, with a given type and excuse. The static method finds the `Schedulable` for the running thread, so you do not need to do any bookkeeping your self. This call is cheap and should be used as close to the blocking region as possible.

`void beginWait(String excuse)`
: Sets status type to `WAIT`

`void beginFileIO(String excuse)`
: Sets status type to `FILEIO`

`void beginNetIO(String excuse)`
: Sets status type to `NETIO`

`void endBlocking()`
: Call this method when leaving the blocking region

Here is an example usage:

```
try{
 SchedulableStatus.beginNetIO("RMI reference decode");
 object = RMIRemoteObjectDecoder.decode(ref);
 } catch (Throwable ex) {
   loggingService.error("Can't decode URI " +ref, ex);
 } finally {
  SchedulableStatus.endBlocking();
 }
```

Cougaar

Cougaar Agent Architecture Open-Source site

ThreadControlService

org.cougaar.core.service.ThreadControlService

`void setDefaultLane(int lane)`

Sets the default lane for any newly created `Schedulables`. This has no effect on extant `Schedulables`.

`void setMaxRunningThreadCount(int count, int lane)`

Sets the maximum number of `Schedulables`that are allowed to run at any one time (across all levels) for the given lane.

`void setQueueComparator(Comparator comparator, int lane)`

Sets the `Comparator` used by the queue at this level for the given lane to order its elements (`Schedulables`). The 'smallest' value, as determined by the `Comparator`, is the first element of the queue. By default the queue is sorted by time (fifo).

`void setRightsSelector(RightsSelector selector, int lane)`

By default, the "right" to run is handled in a round-robin fashion. This can be overridden by providing a different `RightsSelector`. The rights-selection mechanism is experimental and lightly tested, and shouldn't be used (yet) except for experimenting.

`boolean setQualifier(UnaryPredicate predicate, int lane)`

Sets the qualifier at this level for the given lane. The qualifier is used to disqualify queued `Schedulables` temorarily. Any `Schedulable` on the queue which doesn't satisfy the predicate is removed and held in another list. Such a `Schedulable` will only be returned to the queue (and thus given the opportunity to run eventually) if the qualifier is unset (ie, set to `null`).NB: The qualifier can only be set to a non-`null` value if it's current `null` .

`boolean setChildQualifier(UnaryPredicate predicate, int lane)`

Sets the qualifier for child schedulers at this level for the given lane. The qualifier is used to prevent children from gaining rights they might otherwise have access to. This is useful to keep one child from using up all its parents rights (for example).NB: The qualifier can only be set to a non-`null` value if it's current `null`.

`void setMaxRunningThreadCount(int count)`

Sets the maximum number of `Schedulables`that are allowed to run at any one time (across all levels) for the default lane.

`void setQueueComparator(Comparator comparator)`

Sets the `Comparator` used by the queue at this level for the default lane to order its elements (`Schedulables`). The 'smallest' value, as determined by the `Comparator`, is the first element of the queue. By default the queue is sorted by time (fifo).

`void setRightsSelector(RightsSelector selector)`

By default, the "right" to run is handled in a round-robin fashion. This can be overridden by providing a different `RightsSelector`. The rights-selection mechanism is experimental and lightly tested, and shouldn't be used (yet) except for experimenting.

`boolean setQualifier(UnaryPredicate predicate)`

Sets the qualifier at this level for the default lane. The qualifier is used to disqualify queued `Schedulables` temorarily. Any `Schedulable` on the queue which doesn't satisfy the predicate is removed and held in another list. Such a `Schedulable` will only be returned to the queue (and thus given the opportunity to run eventually) if the qualifier is unset (ie, set to `null`).NB: The qualifier can only be set to a non-`null` value if it's current `null`.

`boolean setChildQualifier(UnaryPredicate predicate)`

Sets the qualifier for child schedulers at this level for the default lane. The qualifier is used to prevent children from gaining rights they might otherwise have access to. This is useful to keep one child from using up all its parents rights (for example).NB: The qualifier can only be set to a non-`null` value if it's current `null`.

`int getDefaultLane()`

Returns the current default lane.

`int runningThreadCount(int lane)`

Returns the number of `Schedulables` that are currently running at this level for the given lane.

`int pendingThreadCount(int lane)`

Returns the number of `Schedulables` that are currently queued at this level for the given lane.

`int activeThreadCount(int lane)`

Returns the number of `Schedulables` that are currently either running or queued at this level for the given lane.

`int maxRunningThreadCount(int lane)`

Returns the maximum number of `Schedulables` that are allowed to run at any one time (across all levels) for the given lane.

`int runningThreadCount()`

Returns the number of `Schedulables` that are currently running at this level for the default lane.

`int pendingThreadCount()`

Returns the number of `Schedulables` that are currently queued at this level for the default lane.

`int activeThreadCount()`

Returns the number of `Schedulables` that are currently either running or queued at this level for the default lane.

`int maxRunningThreadCount()`

Returns the maximum number of `Schedulables` that are allowed to run at any one time (across all levels) for the default lane.

Cougaar

Cougaar Agent Architecture Open-Source site

ThreadListener

org.cougaar.core.thread.ThreadListener

`void threadQueued(Schedulable schedulable, Object consumer)`

Subscribed `ThreadListeners` at any given level will receive this callback when any `Schedulable` is placed on the queue of pending 'threads' for that level. The consumer on whose behalf the `Schedulable` was created is also provided.

`void threadDequeued(Schedulable schedulable, Object consumer)`

Subscribed `ThreadListeners` at any given level will receive this callback when any `Schedulable` is popped of the queue of pending 'threads' for that level, in preparation for being run. The consumer on whose behalf the `Schedulable` was created is also provided.

`void threadStarted(Schedulable schedulable, Object consumer`

Subscribed `ThreadListeners` at any given level will receive this callback when any `Schedulable` at that level starts running. The consumer on whose behalf the `Schedulable` was created is also provided.

`void threadStopped(Schedulable schedulable, Object consumer)`

Subscribed `ThreadListeners` at any given level will receive this callback when any `Schedulable` at that level stops running. The consumer on whose behalf the `Schedulable` was created is also provided.

`void rightGiven(String consumer);`

Subscribed `ThreadListeners` at any given level will receive this callback when the next higher Thread Services have granted this level a "right" for the given thread consumer. No `Schedulable` has claimed the right at this point so none is passed in the callback.

`void rightReturned(String consumer)`

Subscribed `ThreadListeners` at any given level will receive this callback when that level returns a "right" to the next higher level. The `Schedulable` which used the right is already back in the thread pool at this point, so it isn't passed in the callback.

ThreadListenerService

org.cougaar.core.service.ThreadListenerService

`void addListener(ThreadListener listener, int lane)`

Adds the given listener to the list of those which will receive callback notifications for ‘interesting’ thread events (queued, dequeued, started, stopped, etc) at this level of the Thread Services, for the given lane.

`void removeListener(ThreadListener listener, int lane)`

Removes the given listener from the list of those which will receive callback notifications for ‘interesting’ thread events (queued, dequeued, started, stopped, etc) at this level of the Thread Services, for the given lane.

`void addListener(ThreadListener listener)`

Adds the given listener to the list of those which will receive callback notifications for ‘interesting’ thread events (queued, dequeued, started, stopped, etc) at this level of the Thread Services, for the default lane.

`void removeListener(ThreadListener listener)`

Removes the given listener from the list of those which will receive callback notifications for ‘interesting’ thread events (queued, dequeued, started, stopped, etc) at this level of the Thread Services, for the default lane.

Cougaar

Cougaar Agent Architecture Open-Source site

ThreadService

org.cougaar.core.service.ThreadService

`WILL_BLOCK_LANE`

This constant refers to the lane that should be used by threads which are known to block on io. MTS Destination Queue threads currently use this lane.

`CPU_INTENSE_LANE`

This constant refers to the lane that should be used by threads which are known to run unusually long [?]. This lane isn't currently in use.

`WELL_BEHAVED_LANE`

This constant refers to the lane that should be used by threads which are known to behave properly, ie not to block and not to run unusually long. The Metric Service threads use this lane.

`BEST_EFFORT_LANE`

This constant refers to the lane which should be used by all other threads. This is the initial default lane (for any given Thread Service, the default lane can also be changed dynamically).

`Schedulable getThread(Object consumer, Runnable runnable, String name, int lane)`

Creates a `Schedulable` with the given name and running the given `Runnable` on behalf of the given consumer (and `Object`). The `Schedulable` will be assigned to the given lane.

`Schedulable getThread(Object consumer, Runnable runnable, String name)`

Creates a `Schedulable` with the given name and running the given `Runnable` on behalf of the given consumer (and `Object`). The `Schedulable` will be assigned to this `Thread Service's` default lane.

`Schedulable getThread(Object consumer, Runnable runnable)`

Creates an anonymous `Schedulable` running the given `Runnable` on behalf of the given consumer (and `Object`). The `Schedulable` will be assigned to this `Thread Service's` default lane.

`void schedule(TimerTask task, long delay)`

This is the equivalent of the corresponding call on `java.util.Timer`. In fact the current implementation of the Thread Services creates a Timer at each layer and uses that Timer in this method, as well as in the next two methods.**THIS METHOD IS DEPRECATED**. Use the corresponding `schedule` on `Schedulable` instead.

`void schedule(TimerTask task, long delay, long interval)`

As above, this is the equivalent of the corresponding call on `java.util.Timer`.**THIS METHOD IS DEPRECATED**. Use the corresponding `schedule` on `Schedulable` instead.

`void scheduleAtFixedRate(TimerTask task, long delay, long interval)`

As above, this is the equivalent of the corresponding call on `java.util.Timer`.**THIS METHOD IS DEPRECATED**. Use the corresponding `schedule` on `Schedulable` instead.

Cougaar

Cougaar Agent Architecture Open-Source site

Use Cases

Use Cases and Examples

Code strips: avoiding calls to wait()

As explained in the overview, `Schedulables` run more effectively as strips of code than as loops with a `wait`, since the blocking behavior of the `wait` will tie up a limited resource needlessly.

As an example, consider a thread-based queue pattern, which might look something the following.

```
class QueueRunner implements Runnable
{
    private SimpleQueue queue;
    private Thread thread;

    QueueRunner() {
        queue = new SimpleQueue(); // make the internal queue
        thread = new Thread(this, "My Queue");
        thread.start();  // start the thread
    }

    public void run() {
        Object next = null;
        while (true) {
            while (true) {
                synchronized (queue) {
                    if (!queue.isEmpty()) {
                        next = queue.pop();
                        break; // process the next item
                    }
                    // queue is empty, wait to be notified of new items
                    try { queue.wait(); }
                    catch (InterruptedException ex) {}
                }
            }
            processNext(next);
        }
    }

    public void add(Object x) {
        synchronized (queue) {
            queue.add(x);
            queue.notify(); // wake up the wait()
        }
    }
}
```

This thread will spend most of its time in the `wait` call, which is not at all an effective use of a limited resource. As a Schedulable it would be better written as follows:

```
class QueueRunner implements Runnable
{
    private SimpleQueue queue;
    private Schedulable schedulable;

    QueueRunner() {
        queue = new SimpleQueue();
        // Create the Schedulable but don't start it yet.
        schedulable = threadService.getThread(this, this, "MyQueue");
    }

    public void run() {
        // Handle all items currently queued but never block
        Object next = null;
        while (true) {
            synchronized (queue) {
                if (queue.isEmpty()) break; // done for now
                next = queue.pop();
            }

            processNext(next);
        }
    }

    void add(Object next) {
        synchronized (queue) {
            queue.add(next);
        }
        // Restart the schedulable if it's not currently running.
        schedulable.start();
    }
}
```

Code strips: avoiding calls to sleep()

Another popular Java Thread pattern uses a sleep in a never-ending loop. As above, this uses up a thread resource even though the thread is rarely running.

```
class SleepingPoller implements Runnable
{
    private int period;
    private String name;

    SleepingPoller(String name, Object client, int period) {
        this.name = name;
        this.period = period;
        ThreadService ts = (ThreadService)
            sb.getService(this, ThreadService.class, null);
        ts.getThread(client, this, name).start();
        sb.releaseService(this, ThreadService.class, ts);
    }

    void initialize() {
        // Initialization code here
    }

    void executeBody()
        throws Exception
    {
        // Thread body here.
    }

    public void run() {
        initialize();
        while (true) {
            try {
                executeBody();
            }
            catch (Exception ex) {
                log.error("Error in thread " + name, ex);
            }

            try { Thread.sleep(period); }
            catch { InterruptedExecption (ex) }
        }
    }
}
```

In this case the use of sleep can be replaced by having the run() method restart the thread after a delay.

```
class Poller implements Runnable
{
```

```
{
    private Schedulable schedulable;
    private boolean initialized = false;
    private int period;
    private String name;
    private ThreadService ts;

    Poller(String name, Object client, int period) {
        this.name = name;
        this.period = period;
        this.ts = (ThreadService)
            sb.getService(this, ThreadService.class, null);
        this.schedulable = ts.getThread(client, this, name);

        schedulable.start();
    }

    void ensureInitiolized() {
        synchronized (this) {
            if (initialized) return;
            initialized = true;
        }

        initialize();
    }

    void initialize() {
        // Initialization code here
    }

    void executeBody()
        throws Exception
    {
        // Thread body here.
    }

    public void run() {
        ensureInitiolized();
        try {
            executeBody();
        }
        catch (Exception ex) {
            log.error("Error in thread " + name, ex);
        }

        // Restart after period ms
        schedulable.schedule(period);
    }
}
```

Code strips: avoiding TimerTasks

`TimerTasks` are not controllable and should generally be avoided. Instead use the `schedule` methods on `Schedulable` for equivalent functionality. In this case the body of the task will run as a COUGAAR thread. Compare `runTask()` and `runThreadPeriodically()` below.

```
class MyPeriodicCode
{
    private int period;
    private Schedulable schedulable;

    public void body() {
        // The body of code to be run periodically goes here.
    }

    // In this version body() runs in the Thread Service's Timer
    // thread, which can be problematic if it takes too long.
    void runTask() {
        ThreadService ts = sb.getService(this, ThreadService.class, null);

        TimerTask task = new TimerTask() {
                public void run() {
                    body();
                }
            };
        ts.schedule(task, 0, period);

        sb.releaseService(this, ThreadService.class, ts);
   }

    // In this version body() runs periodically as a cougaar thread.
    void runThreadPeriodically() {
        ThreadService ts = sb.getService(this, ThreadService.class, null);

        Runnable code = new Runnable () {
                public void run() {
                    body();
                }
            };
        schedulable = ts.getThread(this, code, "MyPeriodicCode");

        // Restart the Schedulable every period ms
        schedulable.schedule(0, period);

        sb.releaseService(this, ThreadService.class, ts);
    }
}
```