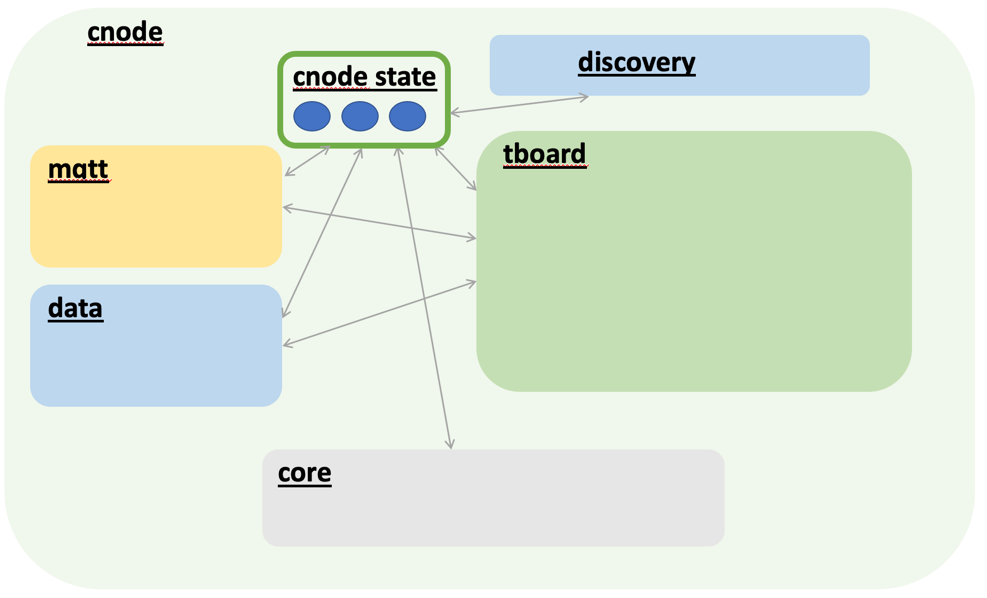
**JAMScript J Controller Design**

**JAMScript Background**

JAMScript is a programming language developed for creating real-time artificial intelligence applications for edge computing systems where rapid mobility (e.g., vehicular mobility) is the norm. You can see more information on JAMScript design in this document. You don’t need to understand the overall architecture to follow the design presented in this document.

**JAMScript C Runtime Architecture**

The JAMScript language runtime has two sides: C and JavaScript. The C side of the runtime is shown in the figure below. The central component of the C runtime is the cnode. There are four major components inside the cnode: mqtt, data, tboard, and core. The **mqtt** is responsible for interfacing with the MQTT brokers and all the remote calls go through this component. There are two types of remote calls: incoming and outgoing. Also, at a given time, a worker can be connected to multiple controllers at different levels. Among these remote controllers (the controllers are represented by their MQTT brokers – that is we connect to the MQTT broker corresponding to the remote controller) we can have one device-level controller, many edge-level controllers and one cloud-level controller.



The **data** component is responsible for pushing data updates from the worker to other workers or the controllers and different levels. In addition to pushing data, the data component can receive updates from the controllers and other workers. The data component has a fast path and slow path. The fast path is the data dissemination among the workers (horizontal dissemination using V2V links in a vehicular situation). The slow path is data pushing between the different levels. For instance, devices pushing data to the edge servers – workers pushing to the edge-level controllers. I will discuss more details about this in the following sections.

The **core** is responsible for maintaining some foundational parameters of cnode. For example, each worker needs to have a unique ID, which is created the first time the node is executed. The core is responsible for maintaining this unique ID. The ID is stored in a file and when the node is run the next time, it is read from the file system. The core reads and writes the configuration files from a well-known location on the file system.

Additionally, cnode maintains shared data structures that are updated by different modules – this shared data structure is shown as **cnode state** in the above diagram. The cnode state is updated by all four components and also is a way for the components to interchange information among themselves.

**The JAMScript Protocol**

The worker (cnode) interchanges messages according to the JAMScript protocol with the controller. Here I explain the JAMScript protocol.

*Worker Discovering a Controller*

Workers expect at least one device-level controller to be in the local network. If there are multiple controllers in the local network, the workers need to have a way of selecting the controller to join. We can have two ways of resolving the problem: discovery request specifying which controller to select or the discovery protocol selecting the closest controller. We can find the closest controller using UDP ping from the worker. The round-trip times for the UDP ping will give us the latency of reaching the controllers.

**Worker >> WHERE-IS-CTRL [g] >> controller**

**<< HERE-IS-CTRL x for g <<**

In the above protocol, we discover the MQTT broker values (IP and port) for the controller. The IP and port can be used to access the broker for later registration messages. The g is a *group name* for the controller. The above protocol uses **UDP multicast** because we haven’t connected to the MQTT when this protocol is run.

*Worker Registering with the Controller*

We want a controller to maintain an accurate knowledge about the set of workers that are engaged with it. This can be used by the controller to distribute work or even accept tasks that have specific requirements on the number of workers required to solve it. The registration protocol is implemented between the workers and the device-level controller. The device-level controller can notify the number of workers to the edge-level controller.

**Worker >> REGISTER id >> controller** *"/app/requests/up"*

**<< REGISTER-ACK <<** *"/app/replies/down"*

Use the topic *"/app/requests/up"* @ the controller's MQTT server (device MQTT only). Use the topic "/app/replies/down" @ the controller's MQTT server (device MQTT only). The controller would keep the registered set of workers. This set is maintained at the controller using the PING protocol that is described below.

**Worker << PING x << controller** *"/app/announce/down"*

**>> PONG id >> controller** *"/app/replies/up"*

Use the topic "/app/announce/down" @ the controller's MQTT server (device MQTT only). Checking the presence of the worker. That is, the controller is asking “I am controller *x,* are you still there?”. The worker would reply with its own ID if it were registered with the controller. So, the worker needs to maintain the information about the registered controller – in this case remember *x*.

*Mobility and Controller to Worker Association*

The workers can move across controllers! The workers are attached to a device-level controller and that association does not change frequently. If a worker finds that the device-level controller it is registered with has failed or not responsive, the worker will go into discovery mode and find another controller. In the case, the worker is unable to find suitable controller to register the worker will terminate. The worker to edge (fog) level controller and cloud-level controller associations can change very frequently. These associations are temporary. The worker learns about edge and cloud level controller through the device-level controller and the following protocol is used to manage this information. In the new design, a worker can be associated with **multiple** edge level controllers.

To get information about cloud or fog (edge) controllers, a worker can send the following message.

**Worker >> GET-CLOUD-FOG-INFO id v >> controller** *"/app/requests/up"*

**<< PUT-CLOUD-FOG-INFO info << controller** *"/app/replies/down"*

The GET-CLOUD-FOG-INFO is giving the worker’s ID and version number and asking the controller for more information about other fog and cloud level controllers. The version number indicates the current version of the information the worker has. The version will be 0 at the startup – no information at the worker. The device-level controller can have new information when the GET request arrives in the controller. In the case the controller has new information the version number at the controller should be higher than the version number sent by the worker. The controller uses the PUT-CLOUD-FOG-INFO message to push the information if it has something new. The controller can send information down without the worker asking for it as well.

*Overall Schedule Management at the Worker*

The task executions at the worker are carried out according to a schedule. This allows precise time-based execution. That is, we can arrange the execution of the tasks to start at precise time points. In order to do this precise task scheduling, we do a schedule computation at the controller and download the schedule to the worker side in a periodic manner. Of course, this schedule downloading is done in a pipelined manner so that the worker does not wait for the schedule to be computed. Before the worker can start the execution of a task, we want to get the schedule downloaded to the worker.

When the workers start, they don't have a schedule. The workers can launch a request to download a schedule and proceed with the execution using a default schedule. The default schedule will have a single partition that using first come first serve (FCFS) as the only policy. The problem with FCFS as a policy is that we don’t know how treat special classes of tasks such as real time tasks or synchronized tasks. So, we need to download a schedule as soon as possible.

A JAMScript program is a directed acyclic graph (DAG) with nodes as tasks and the edges as precedence relations or calls among the tasks. To perform the schedule, we need to know the set of tasks that are going to get executed at any given time. That is, we need to know where in the DAG we are going to be executing next. This is like the branch prediction problem in a normal program where we want to identify the next basic block that would be executed (most likely branch we would take next).

We do a static analysis of the JAMScript program (at compile time) that inserts milestones into the DAG. The milestones are labeled NOP (no operation) statements we insert into the JAMScript program. We do an analysis of the program with inserted milestones to detect the tasks surrounding each milestone and the transitions we can have among the milestones. We determine a milestone transition matrix (MTM) and the tasks with each milestone. Now, when the program runs, we can find out where we are (i.e., which milestone) running. From the MTM we can also find out which milestones we are likely hit next.

The milestone-to-task (MTT) table is something we can determine statically at compile time, and this can be used without modification at the runtime. At the runtime, we determine the milestone we are reaching and update that to the controller. The controller keeps evaluating the MTM and then uses that to predict the next milestones we can reach and then form one or more schedules we could use with the different milestones.

Now the question is the following. **What is the role of the C side in the milestone management process?** The JAMScript code that is executing in the C side is instrumented to include the milestones. As they are executed the runtime is notified about hitting the milestone. For instance, when the program just passes milestone M56, the runtime gets notified by the JAMScript program that milestone M56 was reached. The user program can invoke something like **reach\_milestone(M56)**. So, the runtime needs to support the reach\_milestone(X) API and that is pretty much what it should do.

**Worker >> REACH\_MILESTONE m, pm, ts >> controller** *"/app/requests/up"*

We send the above message to the to the controller. The m is the milestone we just reached at the worker, pm is the previous milestone, and ts is the timestamp. Using the timestamp (high precision) the controller would be able to determine how quickly we are making progress through the different milestones at the worker.

**Worker >> GET-SCHEDULE m, id >> controller** *"/app/requests/up"*

**<< PUT-SCHEDULE {m, s} << controller** *"/app/replies/down"*

Send a message asking for schedule for milestone m. The message we send to the controller will be forwarded to the scheduler that is associated with the controller. We could send a specific milestone for which we need the schedule or a set of milestones. The PUT message pushes a set of schedules down to the worker. The schedules are tagged with the milestones so the worker can put the most appropriate schedule to use.

In addition to the milestone information, the scheduler needs task runtimes. That is, we need to know how long the different tasks take at runtime. This information is collected by the runtime and periodically pushed to the controller and from there to the scheduler.

**Worker >> PUT-EXEC-STAT {t} >> controller** *"/app/requests/up"*

*Monitoring the Network Performance*

A problem that is very important for effective resource management is node and network monitoring. We want to get an accurate estimate of the network performance (e.g., bandwidth and latency) that exists between the controller and workers. So, the workers send continuous probes to the controllers. The probe just sends a random payload to the controller from the worker and let the controller echo it. The worker can estimate the bandwidth and latency using these probes. We can have this process happening between the workers and the controllers at the different levels. A naïve way of implementing this probing protocol would impose significant overhead on the network. Therefore, we use a library that implements a packet tail-gating technique that is capable of detecting bottleneck link capacity without inundating the

The implementation of the probing protocol between controllers and workers is going to work little **differently from the other protocols**. The probing protocol is implemented using a proxy pattern. The controller and worker have proxies (implemented in separate processes) that do the network performance estimation. So, we would use controller-probe and worker-probe to represent the controller and worker, respectively.

The way it would work is for the probe module to register with the main module – controller and worker. Once the probe is registered, we can initiate the probing process. The probe will use a third-party library that does nondestructive bottleneck bandwidth estimation.

*Worker Executing Tasks at the Controller*

In JAMScript, we have a specialized form of remote procedure call (RPC) that allows the worker to launch a task on the controller and vice-versa. One unique aspect of JAMScript is that the RPC calls are one-to-many. Further, the set of fog servers that receive the calls for remote execution changes with time. The tasks in JAMScript can have lot of attributes. Because JAMScript is compiled by a specially crafted compiler that takes both sides into consideration, there is no need to send any of the special attributes with the call. The call is merely sending information necessary to execute the task: name, ID of the initiating node, sequence number, and arguments to the call. All other attributes such as the conditions attached with the task, type of task, etc. are available at the destination node.

**Worker >> REXEC id, params >> controller** *"/app/requests/up"*

**<< REXEC-ACK id, tval << controller** *"/app/replies/down"*

**>> GET-REXEC-RES id >> controller** *"/app/requests/up"*

**<< REXEC-RES id, res << controller** *"/app/replies/down"*

**<< REXEC-ERR id, err << controller** *"/app/replies/down"*

The steps in blue are optional. If the result is not required for the execution, these steps are not used. The params in the above protocol is defined as follows:

**params = {funcName** – unique string to identify the target task,

**seqNum** – a unique sequence number,

**args** – list of arguments, could be empty,

**nodeId** – this is different from taskId (which is outside the params),

it is unique for the node**}**

You send the REXEC call and expect a reply with REXEC-ACK with the matching id (taskId). If nothing comes within a timeout period, the REXEC call is sent again. We repeat this for a set number of retries and fail if no response. If the task is going to give us a result (return value), get into the next two steps (shown in blue). The REXEC-ACK responds with a timeout value (tval). This value is used to wait and send GET-REXEC-RES if the REXEC-RES if not received by then. We send the GET-REXEC-RES command and get either a REXEC-RES or REXEC-ACK. In the first case, we get the result (the task is completed). In the second case, we get a timeout value for which we need to wait. So, the second case implies that the task execution is still in progress. We can also get a REXEC-ERR message that gives the problem (error code) with the remote execution.

We are implementing an “at least once” execution. Note that we are not seeking “exactly once” execution which would be much harder to realize in a transient edge computing environment. The worker launching the call can require a result for the call. If the result is not received within a timeout period, the worker will relaunch the execution call. In the meantime, the fog association could have changed. So, if the result of execution from the fog is lost, we need to repeat the execution in the new fog because the worker wants the results. Therefore, we are implementing the “at least once” execution. The execution is determined by the availability of the results. Because many workers are launching the request for task execution, the controller need not be executing the task for each call. Depending on what the task is doing, we can reuse a previous execution and provide the result to another worker.

*Controller Executing Tasks at the Worker*

Now we consider a controller executing tasks at the workers. There can be many workers underneath a controller, so the call for execution must be executed by many workers. This is like the workers executing on controllers because we can have multiple controllers connected to a worker. However, the number of workers underneath a controller could be quite large compared to the number of controllers a worker could have. There is another difference between the two that is quite interesting. Unlike the controllers, the workers are called upon to execute tasks that can grouped as one of the following types: batch, real-time, or synchronous. That means, the call coming from the controller can be for executing a task that can be any one of the above types. Obviously, the way the task execution must be handled depends on task type.

**Worker << REXEC id, params << controller** *"/app/requests/down"*

**>> REXEC-ACK id, tval >> controller** *"/app/replies/up"*

**<< GET-REXEC-RES id << controller** *"/app/requests/down"*

**>> REXEC-RES id, res >> controller** *"/app/replies/up"*

**>> REXEC-ERR id, err >> controller** *"/app/replies/up"*

The steps in blue are optional. If the result is not required for the execution, these steps are not used. The params in the above protocol is defined as follows:

**params = {funcName** – unique string to identify the target task,

**seqNum** – a unique sequence number,

**args** – list of arguments, could be empty,

**nodeId** – this is different from taskId (which is outside the params),

it is unique for the node**}**

The above description is very much like what we have from the worker to controller. One of the changes you can observe is the topics that are used for doing the publish-subscribe through the MQTT broker.

Now, let’s discuss the different ways the task could be executed at the worker in response to the incoming call for execution. The task could a batch task. In that case, there is no special handling. We just run the task at the worker. If the task is a real-time task, we need to find the slot in the schedule that is meant for real-time tasks. If there are multiple real-time tasks, the slots would be tagged with a label (based on the function name) that can be used to identify the slot to run the task. Same way the synchronous task needs its own slot. In the subsequent sections, we will discuss how the slots are created and enforced.

**A Detailed Look at Cnode**

Cnode is the primary component that encapsulates all the functions in the C side. The JAMScript application written by the user is compiled with calls to this component. I describe the structure of Cnode and how it could be called from the JAMScript application – in the compiled version.

The Cnode component has three major functions. The first one is **cnode\_init()** which allocates the cnode structure and initializes the key functions that go into the structure. The **cnode\_destroy()** is responsible for stopping all services started by cnode\_init() and deallocating all memory allocations. The **cnode\_start()** is responsible for starting the services that are part of cnode.

**cnode\_init(argc, argv) {**

**cn->args = process\_args(argc, argv)**

**if cn->args == NULL terminate\_error(“invalid command line args”)**

**cn->core = core\_init(find\_my\_ip(), cn->args)**

**if cn->core == NULL terminate\_error(“cannot create the core”)**

**cn->devjinfo = find\_dev\_j(cn->core->group)**

**if cn->devjinfo == NULL terminate\_error(“cannot find controller”)**

**cn->devjserv = connect\_mqtt(cn->devjinfo)**

**subscribe\_mqtt(cn->devjserv, cn->core)**

**cn->tboard = tboard\_init(…)**

**}**

Let’s discuss the above pseudo code for cnode\_init() function. We pass the command line arguments to cnode\_init(). First step is to process these arguments and validate the arguments. We terminate with an error if the arguments are not correct. After processing the arguments, we initialize the core. The core needs the IP address of the worker node. A helper function find\_my\_ip() is used for that purpose.

We find the device-level controller as the next step. The find\_dev\_j() function would do a UDP multicast and find the IP and port number of the MQTT broker for the device-level controller. We can optionally specify the group information for the device-level controller discovery. If we are unable to find the controller, the worker would terminate with an error. If we are able to find the MQTT broker, then we initiate the connection. We could do more error checking for this and the following steps in the pseudo code.

**cnode\_destroy(cn) {**

**tboard\_destroy(cn->tboard)**

**core\_destroy(cn->core)**

**destroy\_mqtt(cn->devjserv)**

**for all edge**

**destroy\_mqtt(edge->serv)**

**destroy\_mqtt(cn->cloudserv)**

**}**

The above pseudo code shows the cnode\_destroy() function. It is mainly responsible stopping and deallocating all the components that are started by cnode\_init().

**cnode\_start(cn) {**

**start\_mqtt() ?? is this necessary .. right now mqtt is started automatically**

**tboard\_start(cn->tboard)**

**}**

The cnode\_start() is quite simple. At this point, we have tboard\_start() for sure. We can also start MQTT service so that incoming message are pulled in only this function is executed.

**A Detailed Look at Core**

The Core is responsible for maintaining persistent information about the Cnode. For instance, the nodeID would be stored on the file system and restored at rerun. We can also save other configuration parameter like MQTT port number, serial number, etc. into the structure created by this program.

Because the Core is pretty straightforward, I am not going to explains it operation – actually there is nothing to explain!

**A Detailed Look at MQTT**

This component is pretty complete. So, I will defer its explanation to a later revision of this document.

**A Detailed Look at Data**

This component is one of the newest and involved. The Version 1 of JAMScript has a data component, but Version 2 is much sophisticated. I will explain the data component fully in the next revision of this document.

**A Detailed Look at tBoard**

The Task Board (tBoard) is a central component of the worker runtime. It has many elements:

* Queues that hold the tasks to execute
* Executors that run the tasks from the queues
* Schedules that dictate how the tasks should be executed
* Timing wheel to organize the timeline so tasks get scheduled with precise tolerance

Before we get into a detailed explanation of the tBoard let’s cover some basic concepts like what a task is and so on.

*What is a Task in JAMScript?*

JAMScript is a single-threaded language which uses cooperative multi-threading (or user-level threading) to do multiple tasks. That is, a user-level task scheduler is responsible for scheduling the execution of multiple tasks on one or more kernel-level threads (virtual processors). In the simplest configuration, we use a single kernel-level thread to execute all tasks. When we have multiple kernel-level threads (known as executors) to run the tasks, one executor would be primary and others secondary. To keep the single-threaded nature of JAMScript intact, only **side-effect free** tasks can be **mapped to run on the secondary executors**. We assume that the compiler can tag the tasks as having potential side-effects or not. Using this tagging the task scheduler would map a task to the appropriate executor.

From JAMScript runtime’s point-of-view a task is simply a function that needs execution. In the JAMScript language the programmer would have identified the tasks by prepending a keyword such as ***jtask*** in front of the function definition. Any call to such functions launches a task and that would be submitted to the runtime for execution. The tasks can be called by the local node. That is, a task running in the local node (which is a worker) can launch another task on the node itself. Similarly, we can also have a remote node (only controllers allowed) can launch a task on the worker node. There is not much difference between the two different ways of launching a task: *local launch* versus *remote launch*.

*What are the Different Task Executions?*

A task can return a result or no result. In either case, we run the task and provide a future handle that can be resolved to obtain the outcome of the task. For instance, to get the return result of the task we can call a jwait() on the future. Once the task completes, the jwait() call will return with the result of the task. The above is true for batch tasks. For real time tasks we have a release time and a deadline. The scheduling problem in the worker is hugely helped by the partial schedule that is downloaded from the controller. This partial schedule would have a specific slot for a real-time task or class of real-time tasks. When a real-time task arrives, we are going to launch it to run on the time slot that is marked in the schedule. The schedule is considered *partial* because it does not have a starting time for each batch task. It has a start time for an aggregated batch task where we can map any batch task that may arrive. For real-time tasks we need to put them in the slot that is allocated for them. This is not a very precise scheduling activity. If we manage to start the task on or before the task starts, we should be fine. For the synchronous tasks, the start time needs to be very precise. We can use combination of nano sleeps and busy waits to start the task at the precise time point.

*How are the Tasks Executed?*

The tasks can be executed for a finite amount of time or iterations can go infinitely. In the infinite case, the task will have a while(1) loop or equivalent. That is, the task will iterate through a block of statements for an infinite number of times. To keep multi-tasking (i.e., interleave the operation of different tasks), we need the task to yield its execution at each iteration. Otherwise, we won’t have multi-tasking - only a single task would keep running the whole time. The JAMScript would have inserted the yield statement when it does the compilation.

*Current Implementation of Task Execution in Task Board*

I explain the current implementation by considering an incoming task. That is, we have a request coming into the worker for task execution and let’s see how the execution proceeds. The task is pushed using the *msg\_processor()* function. It is then optionally processed by the sequencer() and then the executor completes it (actually runs the task).

*Proposed Revision to Task Execution in Task Board*

I propose that we split the new additions across the following three files. We add initial processing into processor.c. The msg\_processor() function would do the bulk of the JAMScript protocol processing in processor.c. It is important to note that msg\_processor() function is not in the tBoard. It pushes some of the processing towards the tBoard. Let’s consider PUT\_SCHEDULE and REXEC. The PUT\_SCHEDULE is responsible for installing a schedule in the worker. Suppose there is no schedule yet in the worker. The PUT\_SCHEDULE would install the first schedule.

Suppose the schedule has a length T (measured in microseconds) and it has many slots for executing various types of tasks. For example, at offset x microseconds we have a slot for executing real-time (RT) tasks and similarly other slots. We read the current clock in microseconds and add a startup delay. The startup delay is necessary so that the schedule is going to be feasible. We load the slot events into the timing wheel. Suppose the schedule has three slots: two R and one SY. We know the start times of the slots from the schedule – this is after the start time. We load the start time of the schedule plus the three slots into the timing wheel. The timing wheel has an event: start schedule at S. Another event slot R1 at S + t1, another event slot R2 at S + t2, and yet another event SY at S + t3.

Now at runtime, we start executing tasks. We have the following four cases:

**Case I**: *No RT, SY tasks and No RT, SY slots*: We would not have any events except the start of schedule event on the timing wheel. We get that event and start running the tasks. The tasks are scheduled from the batch (B) queue because there are no tasks in the other two queues. We are using a FCFS algorithm for scheduling the B tasks.

**Case II**: *No RT, SY tasks but RT, SY slots in schedule*: We have RT, SY slots in the schedule. So, we have loaded the timing wheel with all the slots. To optimize the operation, we can monitor the RT and SY queues and if they are empty, we can just keep moving the timing wheel. There is no need to make any comparison of the current time with the timing wheel timed events. We just update the timing wheel – that is move the wheel to the current time point at each task completion.

**Case III**: *RT, SY tasks are there but no RT, SY slots in schedule*: We have RT and SY tasks, but no schedule yet. We can run RT tasks as soon as they come in even if B tasks are waiting. We can ignore the SY tasks that is we dequeue them and not run them. The reason is that SY tasks need synchronization and without a schedule we won’t be able to achieve the synchronization. So, we skip those tasks.

**Case IV**: *RT, SY tasks are there and RT, SY slots in schedule*: Here we have the tasks and the schedule. We need to schedule both types of tasks. I explain the scheduling process in the text below.

The timing wheel holds the future time points (all times in microseconds – absolute times) at which the scheduled time slots would start. So, we execute as task (could be any of the B, RT, SY) and check the situation. We rotate the timing wheel to current time point and two things could happen: we move past an event already in the timing wheel. In that case, the event would be expired now. Alternatively, the rotation of the wheel did not create any expired events. It just made next pending event closer to the current time.

In the case we have an expired event, we need to start that event right away. If the expired event is either RT and SY start slot, we have an error. We log the error and start the task execution immediately. The error report could be fed back to the scheduler or used to suppress running the batch task in the next cycle or both.

In the case of no expired event, we check the time left until the next pending event. If this time is large enough to start a task we start the task. We need to be sure that we can complete the task. A task we start can yield before it completes, so the time we are looking at is the time for the task to yield the executor. If the time to the next pending event is sooner we need to process the RT or SY tasks instead of the normal B tasks.

Suppose the next pending event is an RT task slot (this is the timing wheel event), and we have the RT task already arrived, we just start it. In this case, we could have an early start. With RT tasks, early start is fine. Only late start is a problem because we could have a missed deadline with late start.

Suppose the next pending event in the timing wheel is an SY task slot, and we have the SY task already there in the queue. With SY tasks we cannot launch the tasks right away. We need to wait for the precise time point. We need a precise sleeper which mixes nano sleeps and busy waiting to get to the exact time point (within microseconds). So, we don’t run any task, we just sleep or wait until the SY slot comes up.

We run the SY task and then proceed to the next step.

Now the question is where we would implement all the above mechanisms. I would say we could put most of them in the task\_synchronizer() function. We need to push the tasks into the appropriate queue in msq\_processor() and then process them accordingly in the task\_synchronizer().