Hydra Scheduling Design

This document describes the design of the render farm scheduling system and discusses details about its implementation, including database, core application, and UI aspects.

# Basic Design

The render farm is distributed by nature, with machines acting as autonomous workers rather than being told what to do by an all-powerful whip-cracking overlord. To keep the design consistent, no central scheduler exists in this system. Events are created by farm admins using a UI tool. The events are then stored in a database table which is frequently checked for changes by all of the render nodes. When an event’s time has passed, it is executed once by each applicable render node. If the event is marked as recurring, the first render node to execute it will create a copy of the event to occur one week in the future. Recurring events can be stopped by simply removing them.

# Database Table

The scheduling system requires a new table to be constructed in the primary database. Conceptually, the table looks like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| uid | regexp | date/time | action | recurrence |
| 12345 | 152-.\* | 2013-05-22 08:15:00 | offline | R |
| 12346 | 152-.\* | 2013-05-22 08:30:00 | get off | R |
| … | … | … | … |  |
| 67890 | 152-.\* | 2013-05-23 17:00:00 | online | R |

Additionally, a change must be made to the render node table. Nodes can be in one of two scheduling states: automatic, or manual. A new column will be created to track this state. The actual database table specifications look like this:

RenderNode {

host: string

…

auto: boolean

}

Nodes having FALSE in their ‘auto’ column will not look at the schedule. Nodes having TRUE in their ‘auto’ column will look at the schedule frequently. The user can change the state of the ‘auto’ column at any time using FarmView. This process is described in more detail in the UI section.

Schedule {

eid: long int,

expr: string,

due: date-time,

action: enum,

recur: boolean

}

# Core Application

In order to accommodate the new scheduling system, the structure of the RenderNodeMain service will have to be changed substantially. At present, there is an idle loop which continuously searches for tasks to do, and when it is allowed to do one, it simply starts the task as a sub-process and waits for it to finish. Meanwhile, another thread is running a loop which updates a heartbeat column in the database.

The scheduling system introduces a new query: each render node needs to be looking at the schedule on a consistent basis if it’s in auto mode. Since it’s already doing a heartbeat on a regular interval, there’s no reason it shouldn’t do its scheduling check at the same time. So we see that the heartbeat and schedule check can be combined. If the node is online, it should be looking for tasks. This query could also be rolled into the heartbeat/scheduling transaction as long as it’s done on condition.

Assuming it’s online and not rendering, it should be looking for a task. There is no reason that the node should do these things separately. It should be feasible for each node to do both of those things, and update its heartbeat column, all within a single transaction. Essentially, there will be a database cycle whereby the machine does all of its interaction with the database at once, on a constant interval. If a job is running, the node may simply omit the job-checking query during the database cycle. If auto scheduling is disabled, it may omit the schedule-checking query.

These are the basic queries that each node needs to do during runtime:

1. Pull own info from the render node table for decision-making
2. Update own heartbeat column
3. Hunt for jobs if online and not doing a job already
4. Update own info after accepting or finishing a job
5. Update the task list after accepting or finishing a job

*The database can also be used to allow the user to communicate with the node, although this is sort of a round-about way of doing it if the user is sitting in front of the node. It may be desirable to keep the TCP system solely for messages from localhost-to-self, since that is both a reliable and efficient method of inter-process communication.*

The Get Off command presents some challenges.

Local IPC: if a user presses the Get Off button in FarmView, there should be an immediate response. This can be facilitated using the socket system already in place, so for this case, no changes need be made.

Network IPC: if an administrator needs to send a Get Off command to one or more nodes over the network, TCP is unreliable, and an immediate response is not required. A preferable method would be to pass the command to the nodes using the database. Two possibilities come to mind to make this happen:

1. An event could be created in the schedule which would be executed by the relevant nodes. The problem with this is that it would require all of those nodes to be in auto-scheduling mode in order for them to execute the event. One workaround would involve allowing the administrator to set those nodes back to auto-scheduling mode in order to have them recognize the event. One of the other downsides of using an event is that the administrator would need to write a regular expression for each one.
2. A separate ‘G’ status could be created for render nodes in order to notify them to execute the GetOff command.

# User Interface

FarmView needs to have a new panel to deal with scheduling. This panel is closely related to the render node panel, and so perhaps they can be integrated together somehow. Right now, the render node panel is being shown as a plain table with check boxes and action buttons. It might be convenient to allow users to have a “schedule” button on that same panel. A new dialog could then be opened which would have a regular expression field for selecting machines (which could be easily tested before committing), a field to denote the time of the event, a field to allow selection from a list of possible events, and a check box to enable recurrence. Finally, the user could click Submit or Cancel. Before the event becomes finalized, a basic check is done to make sure that the event will be executed eventually. That is, an error message would be produced if the submitted event would never occur.