v240504.0

`hostess-station` documentation

The `station` module of [`hostess`] implements a distributed task execution and monitoring framework. It is a data flow and process orchestrator that is beefier and more tightly integrated than command-line scripts but much more lightweight than enterprise-scale options like Apache Airflow or dagster.

Note: many of the capitalized words refer to protocol buffer message types (see below). These are generally handled at a high level of abstraction using functions from hostess.station.proto_utils and hostess.station.messages, like dict2msg(), pack_obj(), and make_instruction().

Delegate

A Delegate is a worker/manager object that can perform tasks and gather data for a Station. By default, Delegates do not do anything but send periodic “heartbeats” to their Station. In order to allow a Delegate to perform tasks or gather data, you must attach “elements” – Actors and/or Sensors.

Delegate Main Loop-

When a Delegate initializes itself, it launches a “main” thread, then also launches any attached Sensors (see below) in their own threads. The main thread loops until signaled, performing the following tasks in order:

1. checks if its Sensors have reported actionable events; if so, it pops each of these events off the actionable event queue and sends them to the Station (as PythonObjects embedded in the info field of an Update).
2. checks on the status of its running actions (Actor.execute methods launched in separate threads). If any have crashed or completed, it gathers their results and metadata and sends an Update containing an ActionReport to the Station for each such action.
3. cleans up any completed futures in its ‘threads’ dictionary (generally successful or failed actions)
4. sends a “heartbeat” to the Station if scheduled
5. checks to see if it added Instructions to its queue during any prior communications with the Station; acknowledges and acts on them if so.

Note: Delegate lockout
Although the structure of the main loop should generally prevent race conditions and inappropriate execution, out of an abundance of caution, when communicating with the Station or shutting down, the Delegate “locks” itself. The Delegate skips the following tasks when “locked”:

actionable event checks, action status checks, thread cleanup, scheduled updates, and acting on received Instructions.

Communicating with Station

Delegates use the hostess.station.talkie.stsend() function, a one-shot communication utility that automatically encodes messages as hostess comms, to send Updates to their Station. The content of these Updates may range from “checking in as scheduled” to “here’s your 100 GB file”, but they always include at least basic time and node identification information.

Delegates do not themselves maintain a listening socket. Instead, the Station may choose to use any Report as an opportunity to reply with an Instruction. If the Station has no Instructions for the reporting node, it will reply with a simple acknowledgement string instead. A Delegate adds any Instruction it receives to its instruction_queue list and attempts to interpret it when it next reaches that step of its main loop.

Interpreting Instructions

Delegates interpret instructions differently depending on the Instruction message’s type field (see InstructionType in station.proto). Delegates can currently recognize four types of instructions:

- “do”: The Delegate will check for an Action in this Instruction and attempt to match it against its Actors. If there is no Action or if it doesn’t match the Delegate’s Actors, it will inform the Station that it does not understand the request.
- “config”: The Delegate will look for ConfigParams in this Instruction and attempt to modify its interface to match the requested configuration.
- “stop”, “kill” (same behavior at present): The Delegate will attempt to gracefully shut down.

Termination

If the Delegate receives a shutdown instruction or the main thread encounters an unrecoverable exception, it:

- locks itself
- sets its state to ‘shutdown’ or ‘crashed’ as appropriate
- sends a kill signal to its main loop
- logs that it is beginning shutdown
- detaches itself from its Actors
- attempts to shut down all its Sensors
- clears its queues
- attempts to send an exit report to its Station
- logs its shutdown (and any failures in the shutdown process) locally
- If flagged as the owner of its own process and running inside a launch_node() wrapper, the wrapper will then attempt to terminate the process (with sys.exit())

Station

A Station is a manager object that collects Updates from and sends Instructions to Delegates. Like a Delegate, it does very little by default: it accepts, records, and acknowledges Updates from its Delegates. Attaching Actors to a Station will allow it to respond to changes in system state with useful Instructions. The hostess.station.actors.InstructionFromInfo Actor is an example of a general-purpose utility actor for Stations. You can configure it with rules that will allow the Station to selectively build Instructions for a Delegate when it receives specific Info in an Update.

Stations cannot currently work with Sensors.

Initialization and Main Loop

On initialization, a Station launches a TCPTalk server (see below) in multiple threads. It uses this server to listen for messages from Delegates and send replies back. It also launches its own main loop in a separate thread.

At present, most of the “action” in Station happens asynchronously, triggered by the ackcheck() callback, so at present, the main loop doesn’t do much. On each iteration of the loop, the Station checks to see if any of its server’s threads have crashed, and attempts to relaunch them if so. It also checks its inbox, and if it’s too full, it dumps older messages.

Responding to Updates

When a Station receives a message from a Delegate, it attempts to interpret it as a hostess comm. If it can’t, it sends an “error” response (not fully implemented) to the sender. If the comm contains an ActionReport on a completed task, it logs it and checks to see if any of its Actors can make Instructions in response to it (like for multi-step pipeline management, supplementary database inserts, etc.) Similarly, if it receives Info, it checks to see if any of its Actors can work with that Info. Then, if it has any Instructions queued in that Delegate’s outbox (simply keyed by Delegate name), it formats one as a comm and sends it in response to the Update. Shutdown messages take top priority, followed by config messages, followed by any others. If it doesn’t have any Instructions for the Delegate, it replies with a simple acknowledgement packet.
Important notes about Instruction responses:

- **A new Instruction can be generated within the response cycle; i.e, a Station may reply with an Instruction based on the information it just received.**
- **At present, a Delegate can only understand one Instruction per exchange. This means that if a Station has many queued Instructions for a Delegate, it will have to wait for subsequent heartbeats or completion reports or info messages or whatever to send them all. This is probably ok.**

Termination

Station termination works a lot like Delegate termination, except that it also:

- attempts to send shutdown Instructions to all its associated Delegates, and waits for up to 30 seconds to receive exit reports and ensure successful shutdown
- waits for up to 30 seconds for threads in any locally-running Delegates to complete (to ensure that the process will be able to successfully exit)
- shuts down its TCPTalk server and waits for those threads to complete

Actors and Sensors

Actors

Actors allow *Nodes* to do things. They are modular, and new *Actors* can be freely attached to a *Node* after initialization (using the *.add_element* method). All *Actor* subclasses implement two core methods: *match* and *execute*.

- *match* looks at an instruction and returns True if it looks like the instruction is intended for the Actor – and well-formed – and raises an exception if it does not. This may be any type of exception, although there is a special exception called *NoMatch* that can be used to mark it explicitly.
- *execute* does stuff. What stuff depends on the Actor.

Actors may expose an "interface" consisting of configurable properties. These may be used to modify an Actor's behavior in a variety of ways. For example, the FileWriter class has two interface properties: "file" and "mode". Setting "file" changes the file an instance of that class writes to when it executes; setting "mode" changes the mode it writes in ("a", "wb", etc.) Properties in an Actor's interface attribute propagate up in a "flattened" fashion to the associated Node. This makes it straightforward to send complete configuration instructions to Nodes. For instance, if a Node has an attached FileWriter named "write", the Node will inherit properties "write_file" and "write_mode".

Actors are not intended to be self-managed, i.e., they do not by default have internal constantly-running threads. Every execution of an Actor is directly commanded by a Node. These individual executions run in individual threads in the Node's ThreadPoolExecutor, although they may themselves launch other threads or processes when appropriate (like a FunctionCall actor instructed to launch a daemon in a forked process).

Actors have an important abstract subclass: DispatchActor. This is a type of Actor intended to be attached to Stations to help them construct Instructions for Delegates. It includes a pick() method that can be configured to allow it to mark a specific Delegate or group of Delegates as valid targets for an Instruction.

Sensors

Sensors are partly-autonomous objects designed to monitor events and receive external data. For instance, a Sensor might tail a log file to watch for changes or periodically query a URL. Like Actors, they can also expose a configurable interface, and it propagates to its parent Delegate in the same way.

Each Sensor subclass has a checker() method that defines what it senses and how. Delegates launch each of their attached Sensors in individual threads; these threads loop forever, calling the Sensor's checker method each time.

A Sensor is a little bit like a sub-Node in that it has Actors of its own. These are primarily intended to filter irrelevant events and send reportable ones back to the Delegate to be reported to its Station, but they can also be used for things that do not need to propagate through the Station's network, like cleanup tasks or supplementary logging. Sensors, however, are not full Nodes; they are not capable of managing threads or communicating with the Station. This also means that a Sensor cannot possess Sensors of its own.

TCP server (hostess.talkie.TCPTalk)

This is the lowest-level communication object directly managed by hostess, not counting individual messages. Everything below this is Python's OS-specific raw socket protocol layer.

TCPTalk is a multithreaded TCP server with swappable decoders and receipt callbacks. By default the decoder is read_comm, which decodes messages as hostess comms. Station uses the swappable receipt callback feature to attach a Station method to the server, allowing it to spool messages to and from the Station's inbox and outboxes.

When you launch a TCPTalk instance, it creates a *socket.socket* object and binds it in listening mode to the specified host and port. It then launches a "selector" thread and a configurable number of "i/o" threads. All these threads run (and must run) in the same process, because they communicate with one another via shared Python objects. A single thread can do the job in many cases, but more threads is useful if you are expecting a high volume of incoming messages, especially small ones. The default is 4.

These communication methods include a signals dictionary. As each thread loops, it checks the signals dictionary. If there's any value other than None at the key corresponding to the thread's name, the thread quits; i.e., their main loops look like: "*while self.signals.get(name) is None: ...*" This can be manually modified but is also exposed in the TCPTalk.sig(name) (one thread) and TCPTalk.kill() (all threads + the socket) methods.

The selector thread attaches a basic accept-connection callback (TCPTalk._accept()) to the socket using functionality from the selectors module. Then, it loops forever, checking to see if the selector has queued that or any other actionable event (others would be attached by i/o threads). If it has, it spools the events to i/o threads using the TCPTalk.queues object, which is just a dictionary whose keys correspond to i/o thread names and whose values are lists of tasks for the individual threads. This is all the selector thread does.

The i/o threads loop forever checking for queued events from the selector thread. The expected outcome is that each incoming connection will hit, in order, the *_accept*, *_read*, and *_ack* callbacks. However, these callbacks will usually not all be handled by the same thread. The i/o threads also append each callback as a dictionary to the server's *events* list.

- *_accept()* simply accepts the incoming connection / establishes peering.
- *_read()* reads data from the peered socket and decodes it inline using the TCPTalk instance's decoder() attribute (by default read_comm(), which decodes Hostess comms). It then places the decoded message, along with metadata, into the server's data list.
- *_ack()* acknowledges receipt of message and sends data back to the peer; by default, it sends a simple acknowledgement message, but if the *TCPTalk* instance has its ackcheck attribute set, it will call that function inline to see if additional data is available to return to the peer. Station implements its send-from-outbox functionality by binding the ackcheck() to a Station's TCPTalk server.

General Notes

- all "threads" are Threading.thread objects managed using ThreadPoolExecutors from concurrent.futures. This strategy is highly performant for the kinds of I/O-bound tasks involved in inter-Delegate/Station communication and much more maintainable than asyncio. However, if you anticipate that a single Delegate may simultaneously execute multiple CPU-bound tasks, it is preferable to ensure that their Actors execute them in separate processes. Straightforward functionality exists for this in the FunctionCall actor.
- no autocompiler currently exists for the proto file. If you make changes to station.proto, you must recompile before the package will be able to use them, e.g.: "protoc -I hostess --python_out=hostess hostess/station/proto/station.proto"

Hostess Comms

Hostess comms are the application-layer protocol Stations and Delegates use to communicate with one another. They are designed for use with the TCP interface-layer protocol, which means they are easy to use both locally and over networks.

Packet Format

The lowest level of the hostess comm format – a simple header and footer protocol that help recipients understand, decode, and verify intra-hostess TCP streams. The header format is:

- bytes 0-7: HOSTESS_SOH string (b"\01hostess")
- byte 8: code for message body type (0: none, i.e., raw binary blob; 1: Update protobuf Message; 2: Instruction protobuf Message). In normal usage, most comms will either contain Messages or serve as simple acknowledgement packets.

- bytes 9-12: total length of comm, including header and footer, in bytes (does not support comms larger than 2^32 bytes)

The footer format is:

- bytes 0-7: HOSTESS_EOM string (b"\03hostess")

Protocol Buffer Messages

Major categories:

- Instructions, sent by Stations to Delegates
- Updates, sent by Delegates to Stations
- PythonObjects, packed Python objects with attached deserialization information; can be embedded in many other messages
- Actions, specifications for actions embedded in Instructions
- ConfigParams, configuration specifications embedded in Instructions
- TaskReports, information about a completed or failed task embedded in Updates

Please refer to hostess/station/proto/station.proto for a complete specification of the Protocol Buffer messages.

Note that in most cases you should not have to directly construct protobuf Messages, because the hostess.station.proto_utils and hostess.station.messages modules contain many high-level abstractions for this. For instance, many of these Messages utilize enums for efficiency. It can in some cases be irritating to get the value, rather than key, back out of a protobuf enum field in Python. hostess.station.proto_utils.enum() offers an easy workaround for this.

The hostess.station.messages.Mailbox and hostess.station.messages.Msg classes offer a higher-level interface to viewing and working with Messages. The Station's inbox and outboxes are Mailboxes that automatically display and format Messages as easier-to-work-with Python objects.

Class Hierarchy

- hostess.station.bases.AttrConsumer
    - hostess.station.bases.Matcher *(abstract)*
        - hostess.station.bases.Sensor *(abstract)*
            - various concrete subclasses
        - hostess.station.bases.Node *(abstract)*
            - hostess.station.delegates.Delegate
                - hostess.station.delegates.HeadlessDelegate
            - hostess.station.station.Station
- hostess.station.bases.Actor (*abstract*)
    - hostess.station.bases.DispatchActor(*abstract*)
        - various concrete subclasses
    - various concrete subclasses

- hostess.station.messages.Msg
- hostess.station.messages.Mailbox (*container for Msg but not in inheritance relation)*