pTaskManager: A Broker for Collaborative Autonomy Tasks

Oct 2023

Michael Benjamin, mikerb@mit.edu
Department of Mechanical Engineering
MIT, Cambridge MA 02139
`project-pavlab/appdocs/app_ptaskmgr`

1 Overview

The pTaskManager app is a broker that resides on any vehicle participating in decentralized collaborative decision making with other vehicles.

The task manager does two primary things:

- Accept registrations for alerts from helm behaviors related to bidding on tasks.
- Accept incoming MISSION_TASK messages and spawn helm behaviors related to bidding on tasks.

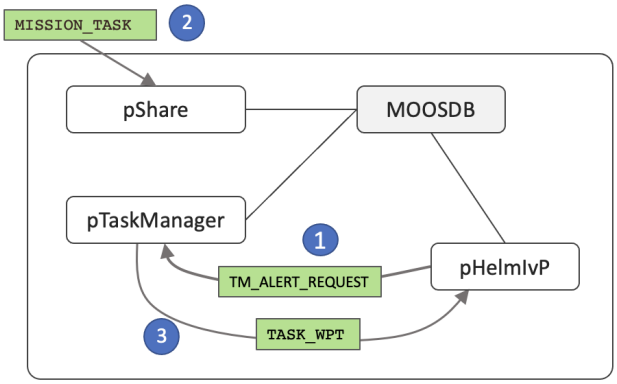

Figure 1: **The pTaskManager Application:** The pTaskManager will (1) accept registrations for mission tasks from helm behaviors related to bidding on tasks. This typically happens when the mission and helm are started. At some point (2) a mission task will be generated and received by the task manager. This may be off-board from a human operator, or from other vehicles, or even from within the same vehicle from another app or behavior. (3) If the incoming mission task type matches a task type known to the task manager, it will generate an alert to the MOOS variable with the format specified in the original alert registration.

The task manager also performs a couple other functions. (a) As a persistent app that is running during the entire mission, it will monitor incoming task bids from other vehicles that may arrive prior to the spawning of a task behavior that needs these bid messages. It retains a cache of "early-arriving" task bids and re-posts them when the task behavior is spawned. (b) The task manager monitors outgoing MISSION_TASK messages and TASK_STATE status messages produced by local task behaviors. It consolidates this information to generate appcasting output which may be useful for debugging and monitoring a mission.

2 The Task Manager Prior to Mission Tasks

The task manager operates as in intermediary between entities producing mission tasks, and entities capable of acting on mission tasks. Entities interesting in acting on mission tasks must register for *alerts*, where each registration specifies:

- The type of mission task
- The name of the MOOS variable that will contain the alert

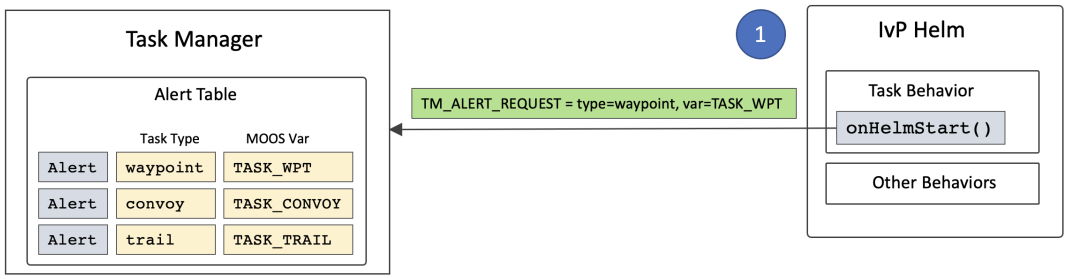

Figure 2: **Alert Registration:** The task manager receives alerts through TM_ALERT_REQUEST messages. The alert request consists of (a) the type of task, and (b) the MOOS variable to be used for an outgoing alert when or if a task of that type is received by the task manager.

Typically the entity that generates the alert request is a task behavior configured within the IvP Helm. The alert requested is typically published in a behavior's `onHelmStart()` method which ensures that the alert request is immediately registered.

Upon the start of the mission, the task manager for each vehicle will produce appcasting output, with a short table like the one below, indicating the registered alerts. The first column shows the task type, the second column is the MOOS variable name that will be used in any future alert related to this task. And the final column shows how many tasks have been received so far for this task.

```
Alerts Registered
=================
TType     VarName      Count
--------  -----------  -----
muster    TASK_MUSTER  0
convoy    TASK_CONVOY  0
waypoint  TASK_WPT     0
```

This table is part of the larger set of appcasting output described in Section 6.

2.1 Timing Considerations for Initial Startup

What happens if the task manager starts after the helm, and the helm generates several alert requests? Given that the MOOSDB only holds the most recent mail for a variable like TM_ALERT_REQUEST, then only the most recent request would be received by the task manager if it started after the helm. The helm has a configuration parameter hold_on_apps which may be configured with a number of apps, such as pTaskManager. The helm will wait to perform its startup functions such as invoking `onHelmStart()` on its behaviors, until all apps in the hold_on_apps list have been detected to be connected to the MOOSDB.

2.2 Timing Considerations for Early Task Bids

It is possible that task bids arrive from other vehicles before a MISSION_TASK message arrives. Bids arrive through the variable TASK_BID. The eventually-spawned task behaviors are perfectly capable of receiving and handling task bid messages. But when bids are received prior to the spawning

of the task behavior, the task manager addresses this issue by also subscribing for task bids. The early bids are cached and then re-posted later when the task manager is sure that a spawned task behavior is ready to receive and handle the bids.

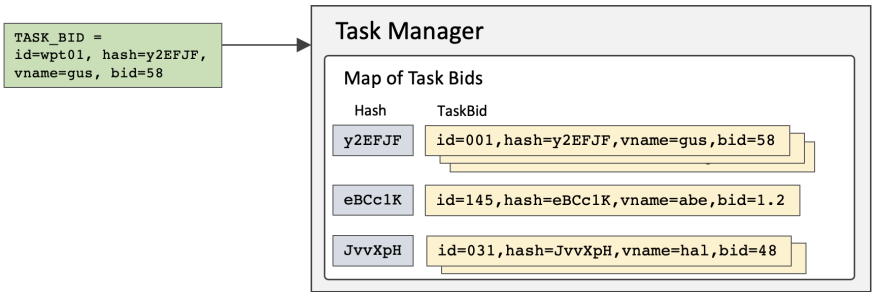

Figure 3: **Early Bid Cache:** The task manager receives task bids and maintains a list of bids associated with each task hash value. These bids are later posted and cleared when the task manager is sure that a task behavior has been spawned and ready to receive the cached bids.

The task manager, on every iteration, will re-publish these cached bids as `TASK_BID` messages, regardless of whether a spawned task behavior exists and is ready to receive bids. Task behaviors are designed to simply ignore duplicate task bids. Task behaviors also produce `TASK_STAT` messages, consumed by the task manager. When the task behavior indicates that bidding is completed, the task manager will clear the early bid cache for that task hash.

3 Task Manager Upon a New Mission Task

The genesis of a new vehicle task is the publication to the MOOS variable `MISSION_TASK`. This may originate (a) off-board from an operator at a command-and-control station, (b) off-board from another vehicle, (c) onboard from another MOOS app, or behavior within the helm. The task manager receives this message and extracts the mission task *type*. If the task type matches one of the previously received alert registrations, an alert will be generated. The alert consists of the MOOS variable identified in the registration. The alert, when received by the helm, will result in the spawning of a new task behavior, which will begin the process of generating a bid for the task and soliciting bids from other vehicles. This process is shown in Figure 4.

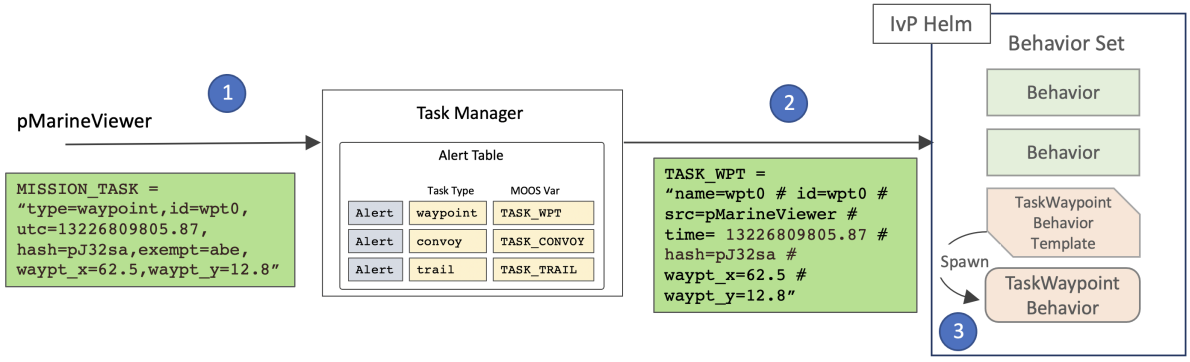

Figure 4: **A New Mission Task:** (1) When the task manager receives a new MISSION_TASK message, it checks to see if the task type is known, from a prior alert registration. (2) If the type is known, a new alert is generated using the MOOS variable specified in the prior alert registration. (3) The alert will result in a new behavior spawning in the helm.

3.1 Mission Task Messages

A *mission task* is specified in a posting to the MISSION_TASK variable. It consists of six components:

- The task type
- The task ID
- The task hash
- The task time-stamp, in UTC seconds
- The task details
- A list of exempt vehicles

type	ID	exemptions	timestamp	hash	details
type=waypoint, id=wpt0, exempt=abe:ben, utc=13226809805.87, hash=pJ32sa, waypt_x=62.5,waypt_y=2.8					
mandatory	mandatory	optional	optional*	optional*	mandatory

Figure 5: **Mission Task Message Structure:** A mission task is posted to the MOOS variable MISSION_TASK. It contains two always-mandatory fields, `type`, and `id`. It contains two optionally-mandatory fields, `utc` and `hash`. The sixth field, `exempt`, is always optional and contains a list of vehicles exempt from this task. The tasks must also contain one or more "details" fields, which specify how to accomplish the task. In this case, since the task is a waypoint task, the x-y components of the waypoint are specified.

The task type and task ID are always mandatory. The task type allows the task manager to match the task with previously registered alerts. The task ID allows the task manager to (a) discern whether two closely arriving and task of the same type represent two different tasks, or are the same task due to a repeated incoming message, and (b) the task manager uses the task ID to form content in the alert message.

The task UTC time-stamp and the task hash are optionally mandatory. They provide further clarity on the uniqueness of the mission task. They are discussed further in Section 3.2.

The *task details* component is specific to the mission. For example, if the task is to transit to a waypoint as the example above, the details component specify where that waypoint resides. If the task were to follow a vehicle, the details will describe the name of the vehicle to follow. Two things to keep in mind: (1) although the content is completely unrestricted, *something* must be specified. Any field not equal to `type` `id`, or `exempt` will be regarded as a details component. (2) The format of the details match the configuration behavior meant to be spawned when a task is received. For example, the `BHV_TaskWaypoint` behavior supports the two configuration parameters `waypt_x` and `waypt_y`.

Mission tasks typically originate off-board, either from a command-and-control station, or another vehicle. These are critical messages and likely full under the realm of mediated messaging, with `pMediator` or similar. As such, the original message may be delayed. It may also be duplicated if the ack for a prior message was dropped. If a `MISSION_TASK` message is repeated, the task manager will discern if it is a duplicate and simply ignore duplicates. The `utc` and `hash` fields provide an extra level of clarity in discerning whether a message is a duplicate.

3.2 Mission Task Hash and UTC Components

An incoming task message may include both a `hash` and a `utc` timestamp component. These components are relatively new additions to the message structure, and for this reason are optional. They are recommended, however, as additional guards against edge-case scenarios where we want to be clear that two task messages refer to two distinct tasks. They can also be helpful for post-mission tools that process logged mission task data.

The `hash` component is typically used when the source of the task is a single entity. In Section 5.1, a mission scenario is described where the task originates from the shore. For example, a task message is sent to a group, asking one vehicle to return to port. In an extreme case, if this message is sent, but not acknowledged, the shore would keep re-sending the message until an acknowledgment was received. In the meanwhile, the group may have actually received the task message, and may then receive the same message again since the shore has repeated the message, possibly resulting in two vehicles returning. This duplicate message can be detected in the `pMediator` app if it is being used. But if the original message contains a unique hash, then the task mananger may also be able to discern between duplicate and unique messages.

The `utc` component can also act as a unique identifier, if it is applied by the source of the mission task message. The time stamp serves as a indicator of the age of the task, which is used for debugging output, and used as a criteria for deleting from memory.

What happens when a `hash` or `utc` field is not provided? If the `utc` field is not present on an incoming `MISSION_TASK` message, then `pTaskManager` will assign a time-stamp for the task based on the observed time upon receipt of the message. If the `hash` field is not provided on an incoming `MISSION_TASK` message, then `pTaskManager` will assign it a hash value equivalent to the task ID field.

By default, `pTaskManager` is configured to allow incoming tasks without `utc` or `hash` fields. However, `pTaskManager` may be configured with the parameters

task_utc_mandatory=true (default is false)
task_hash_mandatory=true (default is false)

These default values were chosen for backward compatibility. It is recommended that they be set to true, and measures be taken in any source application or behavior to generate mission tasks with these fields.

NOTE: The IvP Helm behaviors support macros $[HASH] and $[UTC] to aid in constructing mission task messages from event flags.

3.3 Generating an Alert from a Mission Task

An incoming mission task will generate an alert, as depicted in Figure 4. The alert is constructed to confirm with the IvP helm syntax for a behavior update. In this case the behavior will be a special kind of behavior, a *task behavior*, whose sole purpose is to participate in an auction related to the incoming task. Recall two key rules regarding behavior updates:

- If the name field in the update (alert) message matches a behavior name that already exits, then all the other fields in the update message will be applied to the existing behavior as parameter modifications.
- If the name field in the update (alert) message does NOT match an existing behavior name, the message will result in the spawning of a new behavior with the given name. All other components of the update message will be applied to the new behavior as initial settings.

A behavior that receives an alert and spawns a new behavior is regarded as a behavior *template*. No such instantiated behavior of that type exists upon helm startup until a valid update message has been received. These template behaviors are configured with templating=spawn in their initial configuration.

The contents of the alert nearly match the contents of the incoming mission task message, with the exception of the additional src component. The source is derived by the task mananger by looking at the source field of the incoming MOOS message packet of the MISSION_TASK mail. This is helpful later on for bookkeeping and debugging. If the MISSION_TASK message contains a timestamp in the utc field, this timestamp is passed on in the alert message. Since the utc field is a recommended but not required field, it may not be present in the incoming MISSION_TASK message. In this case the time field in the posted alert will be filled in with the noted time that the task manager receives this message.

3.4 Releasing Cached Bids to New Task Behaviors

When a new task alert has been generated, upon receiving a new MISSION_TASK message, the task manager will begin posting any task bids that may have arrived earlier than the mission task. These task bids are held in a cache as described in Section 2.2. Once a new task has been received, it is considered *active* until completed. On each iteration of the task manager, the cached bids for any active task will be reposted as TASK_BID messages, for the consumption of the newly spawned task behavior.

4 Task Manager After a Mission Task

The task manager plays a small role in a task once it has been received and it has generated the appropriate alert. It essentially does two additional things:

- It monitors the task state, from the initial state of `spawned` up through the bidding process and result. This is primarily so a user can monitor and debug a situation through the pTaskManager appcasting output.
- It monitors and temporarily holds bids coming in from other vehicles, to gaurd against task bids that may arrive before a mission task has been spawned.

These two roles are discussed in this section.

4.1 Task Manager Monitoring of Open and Recent Task Auctions

The task manager, after receiving a mission task and posting an alert, will passively monitor the spawned task behavior progress for a while. It maintains a table of mission tasks, keyed on a task's hash value. This table is rendered in the appcasing output of pTaskManager, similar to the below example:

```
Incoming Tasks: Active(1) Completed(7)
=====================================
ID          Hash       TType   Age  State    TSize  Src           Details
----------  ---------  ------  ---  -------  -----  ------------  ------------
follow_deb  CW5k8I     convoy  2    bidding  3      deb+pHelmIvP  contact=deb
----------  ---------  ------  ---  -------  -----  ------------  ------------
mst_one     mst_one    muster  4    bidlost  4      pHelmIvP      region=one
follow_ben  eRwmw0     convoy  370  bidwon   3      ben+pHelmIvP  contact=ben
mst_two     mst_two    muster  373  bidlost  4      pHelmIvP      region=two
follow_cal  fGKl1h     convoy  535  bidwon   1      cal+pHelmIvP  contact=cal
follow_ben  jcBkSF     convoy  538  bidlost  2      ben+pHelmIvP  contact=ben
follow_deb  cZ4pMp     convoy  542  bidlost  3      deb+pHelmIvP  contact=deb
mst_three   mst_three  muster  543  bidlost  4      pHelmIvP      region=three
```

Most of this information, with the team size (`TSize` column) and the task state, are known to the task manager at the outset upon receiving the incoming mission task message. These other two pieces of information are provided by the spawned task behavior. The task behavior, while it is instantiated and participating in an auction, will periodically post status updates through the TASK_STATE message. for example:

```
  TASK_STATE = id=follow_deb,hash=CPpy1V,state=bidlost, team_size=2
```

The contents of the TASK_STATE message contain the two pieces of information only the task behavior could know: the state and teams size. The other two pieces of information, the ID and hash values, provide clarity on exactly which task this message refers to.

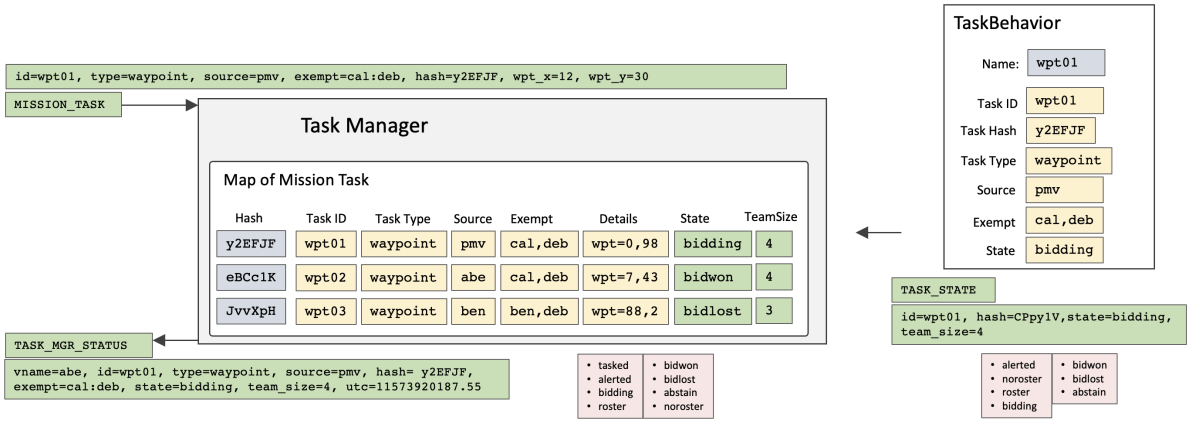

Figure 6: **Active Task Table:** The task manager receives task bids the cached bids.

4.2 Monitoring Task State Updates from Task Behaviors

The information in the task manager table shown in Figure 6 is derived in part from information known about the task at the outset, and information received from task behaviors after they have been spawned. The primary status update output from task behaviors arrives from `TASK_STATE` messages. For example:

```
TASK_STATE = id=wpt01, hash=CPpy1V, state=bidding, team_size=4
```

These messages are matched to the task manager table by the ID and hash values. Each active mission task will hold the current task state and size of the bidding team. The bidding team includes ownship. The possible task states are:

`tasked, alerted, noroster, roster, bidding, bidwon, bidlost, abstain`

The tasks `bidwon`, `bidlost`, and `abstain` are considered *result* states. When a result stage has been achieved, the task mananger will remove the task from the active task table, to a list of recently archived tasks. Once the task has been removed from the active task table, the task manager will no longer re-send any cached task bids associated with this task.

4.3 Summarizing Task Status for the Shoreside

The task manager will publish its own status update, per task, with the idea of sharing this information to the shoreside community. A separate app, `uFldTaskMonitor` allows the operator to monitor tasks across all vehicles through its appcasing output. This variable is `TASK_MGR_STAT`, for example:

```
TASK_MGR_STAT = vname=abe, src=pmv, exempt=cal:deb, id=wpt01, hash=CPpy1V,
                state=bidding, team_size=4, utc=11573920187.55
```

The syntax of this posting is very similar to the `TASK_STATE` messages produced by the task behaviors

and consumed by the task manager as described in Section 4.2. However, it also contains the vehicle name, task source, and timestamp of the original task. The latter three pieces of information are of no concern in reporting state from the task behaviors, since the task manager knows this information already for each task. But this quite useful from the shoreside `uFldTaskMonitor` perspective which may be learning about a new task through such messages.

5 Mission Task Scenarios

Mission tasks can originate and proceed in a variety of ways. We describe here three common scenarios and how they may differ with respect to handling in the task manager.

5.1 Shoreside Operator Generated Tasks

A task may be generated by an operator, with a command-and-control comms connection to a group of vehicles. For example, an operator may task a group of vehicles to return one for fueling, where the group applies a bidding policy that may depend on who needs fueling, who is closest to the fuel point, who is busy doing something else and so on. After the initial message is sent, the task manager on each vehicle receives the task, spawns a task behavior with said policy implemented, and begins to exchange bids with other vehicles as shown in Figure 7.

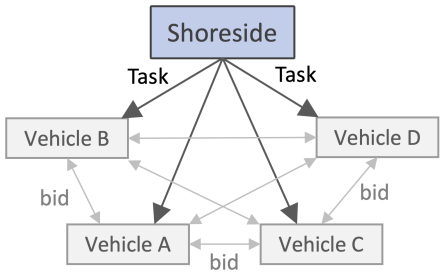

Figure 7: **Operator Generated Tasks:** The operator, typically engaging in the shoreside or topside community through a command-and-control GUI such as `pMarineViewer` or similar, generates a mission task broadcast to all relevant vehicles. The vehicles then generated bids among themselves to determine who should act on the task.

Other use cases may include tasking a vehicle to investigate a certain environmental event such as a plume. In swarm defense scenarios, an operator may task a group to select a vehicle to intercept and investigate a contact that may need further investigation.

5.2 Single Vehicle Collaborator Generated Tasks

A task may also be generated by a vehicle to a group of collaborating vehicles. For example a vehicle may be selected to lead a linear convoy to another location. The selected vehicle may then task the remaining group to be its "number-two", the follower directly behind the leader. The group may apply a policy base on range and relative bearing to the lead vehicle, obstacles between itself and the lead vehicle, or who is busy doing other mission components. After the initial message is sent from the originating vehicles, the task manager on all the receiving collaborator vehicles will

each spawn a task behavior with said policy implemented and begins to exchange bids with other vehicles as shown in Figure 8.

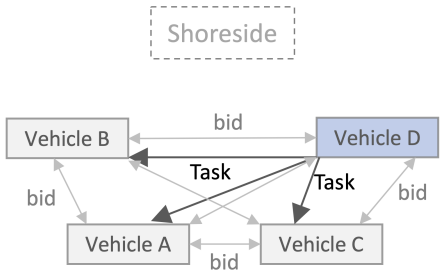

Figure 8: **Collaborator Generated Tasks:** A vehicle, typically part of a co-deployed, collaborative group, generates a task to be sent to other collaborating vehicles in the group. Typically, but no always, the tasking vehicle may exempt itself from the assigned task. The task may be initiated on the generating vehicle any manner that produces a MOOS publication, including event flags from a a behavior.

Using this style of task generation, a set of *cascading* tasks may be generated where each vehicle, upon winning a task, generates a new task for the remaining vehicles. Since tasks may be generated with exemptions, each winning vehicle adds itself to the exemption list until the final vehicle is the sole bidder on the last task. This method is used for linear convoy generation. The first vehicle is chosen as the lead vehicle either explicitly from a commander, or by winning an task auction. It generates a new task to find a number-two follower. The winner of the number-two follower generates a task to be its follower and so on down to the last vehicle in the linear convoy.

5.3 Circumstance Generated Tasks

The task manager supports a distinctly different form task generation, where each vehicle in a group is responsible for tasking itself. In this situation, typically each vehicle is progressing through a course of mission stages, e.g., transiting to a waypoint, or reaching a set-point in a muster behavior. Each vehicle is configured to reach a stage at roughly the same time, or in a relatively brief common window of time. Each vehicle is configured to generate task to itself when it reaches this state. It does not share this task to other vehicles, but rather, waits for the other vehicles to reach their stage and self-generate the same task. As each vehicle self-generates the task it promptly generates bids for the task and sends these bids to the other vehicles. This idea is shown in Figure 9.

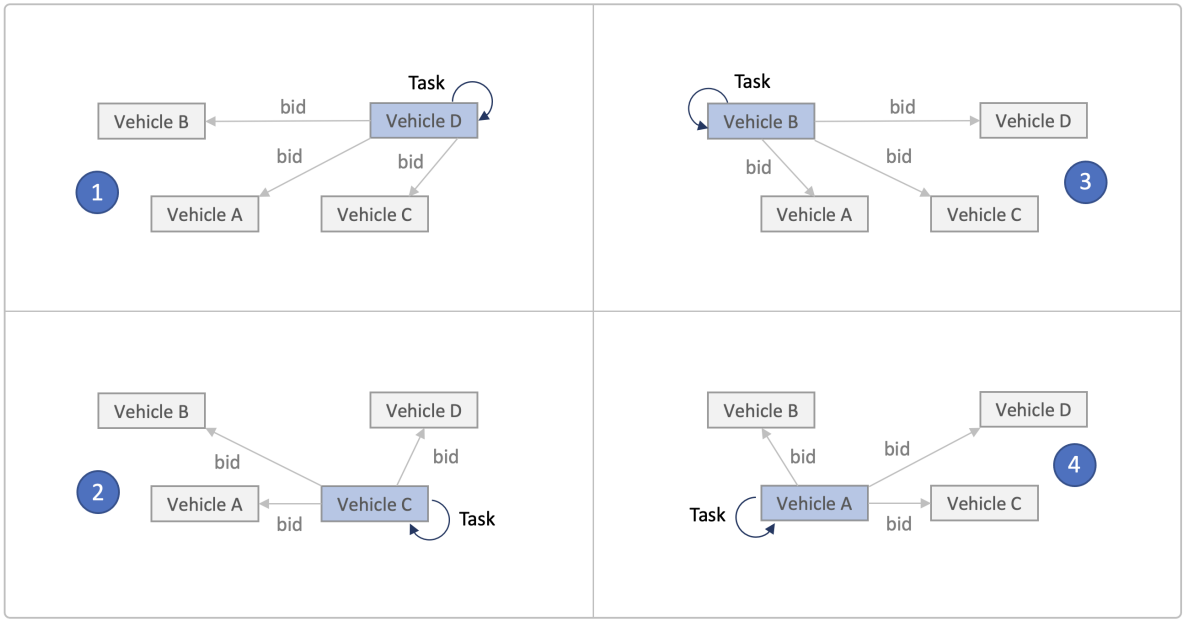

Figure 9: **Circumstance Generated Tasks:** A set of vehicles, proceed on a course of events designed to each reach a stage at roughly the same time, or in a relatively brief window of time. (1) As the first vehicle reaches the stage it generates a task only to itself, triggering bids out to all collaborating vehicles. (2-3) As successive vehicles reach the stage, they also generate tasks to themselves and send out bids. The auction remains partially complete until the final vehicle reaches its stage, also triggering a task to itself. (4) The bids sent from the final vehicle to all other vehicles, will complete the auction on each vehicle thereby completing a consensus on the task.

Since the task generation is happening asynchronously, the early bids will arrive on collaborating vehicles before any relevant task has been generated. As discussed in Section 2.2, the task manager is prepared to receive bids that arrive prior to any corresponding task. These bids are held in the bid cache and released later when a mission task does appear and a corresponding task behavior is ready to receive the bids.

The initial motivation for this style of task generation was to support a mission where a set of vehicles were co-deployed to muster in a given muster region. Upon arrival (each vehicle acquiring its muster set-point), the vehicles generate a convoy task to proceed to the next muster region. The initial convoy task is a circumstance generated task, where the circumstance was muster-region arrival.

6 AppCasting Output for pTaskManager

The AppCasting output for `pTaskManager` is shown below. As with all AppCasting output, the first line identifies the app (`pTaskManager`, the `0/0(3577)` indicates there are no configuration or run warnings, and this is iteration `3577` of this app.

The body of the output is broken into (1) Configure Settings (2) Registered Alerts, (3) Incoming Tasks, (4) Outgoing Tasks, and (5) Most Recent Events.

The *Configuration Settings* section shows the values of the five configuration parameters for this app.

The *Registered Alerts* section shows the three current alerts for the task manager. These are added upon incoming TM_ALERT_REQUEST messages as described in Section 2. The table in this section shows the alert type, the MOOS variable name to be used upon an alert, and the number of incoming mission tasks for each alert type. Note, in this case, there have been 8 total alerts, and if you peek ahead to the next section, the total incoming tasks is also 8.

In the *Incoming Tasks* section, a table is maintained for each incoming task, with the active tasks at the top, and the completed tasks below the separator line. Typically tasks only remain active for a short period of time, and often this first section is empty. Recall that a task is "complete" or "resolved" when it reaches either the `abstain`, `bidwon`, or `bidlost` state. As discussed in Section 4.1, 6 of the 8 columns are known to the task manager from information in the incoming task. The information in the `State` and `TSize` columns comes from incoming TASK_STATE messages from the task behavior spawned for this task. The rows of completed tasks is limited by policy. Only the most recent 8 completed tasks are retained and shown in the table. This limit can be overridden with the completed_task_memory parameter.

The *Outgoing Tasks* section, the task manager shows recent tasks that may have originated by ownship. The task manager essentially snoops on outgoing NODE_MESSAGE_LOCAL messages for any messages containing a MISSION_TASK. No action is taken by the task manager other having this information to include in the AppCasting output for operator situational awareness. Only the most recent 8 outgoing tasks are retained and shown in the table. This limit can be overridden with the outgoing_task_memory parameter.

The final section *Most Recent Events* is standard AppCasting output of events, limited to the 10 most recent events. In the case of the task manager, events include recent incoming bids, alert messages, new task states reported by task behaviors, and outgoing bids.

Listing 6.1: Example pTaskManager AppCast Output.

```
================================================================
pTaskManager abe                                        0/0(3577)
================================================================
Configuration Settings:
=======================
  active_task_memory:    15
  completed_task_memory: 8
  outgoing_task_memory:  8
  task_utc_mandatory:    false
  task_hash_mandatory:   true

Registered Alerts:
======================
TType     VarName      Count
--------  -----------  -----
waypoint  TASK_WPT     0
convoy    TASK_CONVOY  5
muster    TASK_MUSTER  3

Incoming Tasks: Active(1) Completed(7)
====================================
ID          Hash       TType   Age  State    TSize  Src           Details
----------  ---------  ------  ---  -------  -----  ------------  ------------
follow_deb  CW5k8I     convoy  2    bidding  3      deb+pHelmIvP  contact=deb
----------  ---------  ------  ---  -------  -----  ------------  ------------
mst_one     mst_one    muster  4    bidlost  4      pHelmIvP      region=one
```

```
follow_ben  eRwmw0     convoy  370  bidwon   3      ben+pHelmIvP  contact=ben
mst_two     mst_two    muster  373  bidlost  4      pHelmIvP      region=two
follow_cal  fGKl1h     convoy  535  bidwon   1      cal+pHelmIvP  contact=cal
follow_ben  jcBkSF     convoy  538  bidlost  2      ben+pHelmIvP  contact=ben
follow_deb  cZ4pMp     convoy  542  bidlost  3      deb+pHelmIvP  contact=deb
mst_three   mst_three  muster  543  bidlost  4      pHelmIvP      region=three

Outgoing Tasks:
===================================
ID          Hash    TType   Age  SentTo   Exempt   Details
----------  ------  ------  ---  -------  -------  -----------
follow_abe  mKV3yw  convoy  367  cal,deb  abe,ben  contact=abe

======================================================================
Most Recent Events (10):
======================================================================
[973.52]: TASK_MGR_STAT=vname=abe,id=follow_deb,source=deb+pHelmIvP,hash=CW5k8I,state=bidding,
          team_size=3,utc=13231412625.17
[973.51]: New Task State (hash=CW5k8I): bidding
[973.51]: outgoing bid to cal: bid on tid=follow_deb,hash=CW5k8I, bid=83.02
[973.51]: outgoing bid to ben: bid on tid=follow_deb,hash=CW5k8I, bid=83.02
[973.23]: TASK_MGR_STAT=vname=abe,id=follow_deb,source=deb+pHelmIvP,hash=CW5k8I,state=noroster,
          team_size=1,utc=13231412625.17
[973.23]: New Task State (hash=CW5k8I): noroster
[972.95]: TASK_MGR_STAT=vname=abe,id=follow_deb,source=deb+pHelmIvP,hash=CW5k8I,state=alerted,
          team_size=1,utc=13231412625.17
[972.95]: TASK_CONVOY=name=follow_deb # id=follow_deb # src=deb+pHelmIvP # hash=CW5k8I #
          exempt=deb # contact=deb
[972.12]: New Task State (hash=mst_one): bidlost
[971.85]: INCOMING bid from cal: id=mst_one,hash=mst_one,vname=cal,utc=13231412623.838,bid=991.25
```

7 Publications and Subscriptions for pTaskManager

The interface for pContactMgrV20, in terms of publications and subscriptions, is described below. This same information may also be obtained from the terminal with:

```
$ pContactMgrV20 --interface or -i
```

7.1 Variables Published by pTaskManager

The primary output of pTaskManager to the MOOSDB is the set of user-configured alerts and cached bids. Other variables are published on each iteration where a change is detected on its value:

- APPCAST: Contains an appcast report identical to the terminal output. Appcasts are posted only after an appcast request is received from an appcast viewing utility. Section 6.
- TASK_MGR_STAT: A status message, per task, containing the task state. Typically shared to a shoreside application for enabling a collective status view across all vehicles. Section 4.3.
- TASK_BID: Task bids from other vehicles are read and cached by the task mananger for re-posting to spawned behaviors. Section 3.4, and Section 2.2.

The task manager also publishes any user configured alerts.

7.2 Variables Subscribed for by pTaskManager

The pContactMgrV20 application will subscribe for the following MOOS variables:

- APPCAST_REQ: A request to generate and post a new appcast report, with reporting criteria, and expiration.
- MISSION_TASK: A new task, originating externally or from ownship, comes through this mail message. Section 3.
- TASK_STATE: The task behaviors spawned by the task manager produce this message to update the task manager on the progress of each task. Section 4.1.
- TM_ALERT_REQUEST: Task behavior templates will publish to this variable, essentially requesting the task manager to publish an alert when a new mission task of the specified type is received. Section 2.
- NODE_MESSAGE_LOCAL: Outgoing node messages are snooped on to glean outgoing mission tasks and outgoing task bids, solely for AppCasting situational awareness. Section 6.
- NODE_MESSAGE: Incoming node messages are ingested to look for incoming task bids. The task manager will re-publish task bids under the TASK_BID variable. The task manager avoids registering for this variable by gleaning task bids from node messages. Section 3.4, and Section 2.2.

7.3 Configuration Parameters for pTaskManager

The following parameters are defined for pTaskManager. A more detailed description is provided in other parts of this section. Parameters having default values are indicated so.

Listing 7.2: Configuration Parameters for pTaskManager.

task_utc_mandatory:	If true, incoming tasks must have a UTC timestamp. The default is false. Section 3.2.
task_hash_mandatory:	If true, incoming tasks must have a hash field set. The default is true. Section 3.2.
max_active_task_memory:	The maximum number of active tasks held in the task manager memory before auto-deletion based on age. The default is 15. Section 6.
max_completed_task_memory:	The maximum number of completed tasks held in the task manager memory before auto-deletion based on age. The default is 8. Section 6.
max_outgoing_task_memory:	The maximum number of outgoing tasks held in the task manager memory before auto-deletion based on age. The default is 8. Section 6.