MPAM Architecture Overview (Incoherent Version)

Evan Kirshenbaum

Tuesday, March 22, 2022

An introductory apology

This is a first attempt to sketch out how the Thylacine MPAM architecture works. I had hoped to have a couple of weeks free to focus on this before trying to get somebody else up to speed, but that, alas, does not appear to be in the cards, and it's rapidly becoming clear that it is important for me to get *something* down on paper. So, here we are.

In the interest of getting something potentially useful in place before I leave on vacation in a couple of days, this first draft is going to aim at providing at least a shallow sketch of as many of the principal components, classes, and concepts of the system as possible, rather than going into exhaustive detail on the APIs. I'm going to assume that the reader has access to the source code and can look at it for more information. Of course, if what I say here contradicts what's in the code, believe the code.

As I'm writing under time pressure, this will necessarily be more or less stream-of-consciousness. One of the things that has kept me from writing this up to now is that I haven't been able to figure out what the appropriate pedagogical organization is. There are a lot of pieces, and it's not clear where to start. So, I'm just going to … start. Hopefully, by a later draft, a better organization will have become clear, and I can rearrange things appropriately. Until then, you may have to jump around a bit.

Principal components

In this section, I will list what I see as the main conceptual components of the architecture. Each will get its own section later on. Yes, I do realize that this list not only contains apples and oranges, it also contains purple.

- The **system** (page 3), represented by a *System* object, represents the physical hardware of a particular installation. It contains one or more *SystemComponent*s, notably a *Board* and, often, a *Pipettor*. It is also associated with a *Clock*, an *Engine*, and, perhaps, a *Monitor*, all of which will be described below. A *Board*, in turn has *BoardComponent*s, including *Pad*s, *Well*s, *Heater*s, *Magnet*s, and *ExtractionPoint*s (pads via which small amounts of liquid can be extracted or delivered directly). By means of **with** statements, the *System* can also both prevent the engine from terminating and ensure that multiple requests to the *Engine* are submitted atomically.

- The system's **clock** (page 4), represented by a *Clock* object, ticks at a regular rate[1], where that rate can be adjusted by the program and paused and restarted. Actions can be scheduled to take place on a given tick (or the next tick or a certain number of ticks in the future) or immediately before or after any "on tick" actions. The actual implementation of the clock is performed by the system's engine's clock thread.
- The **engine** (page 4), represented by an *Engine* object, contains three background threads: the *ClockThread* performs actions according to the ticking and pausable system clock: the *TimerThread* performs actions delayed by wall-clock time, and the *DevCommThread* performs actions involving communicating with hardware. Each of these *WorkerThread*s, runs its own loop, waiting for requests to be enqueued.
- The **quantities** package (page 4) supports simple type-safe use of dimensioned quantities and units. The dimensions used by MPAM include *Time* (and absolute *Timestamp*), *Volume*, *Temperature* (and absolute *Temperature-Point*), and *Frequency* as well as MPAM-defined *Ticks* and *TickNumber*.
- There are a whole raft of **support types** (page 4) that are used to represent concepts important to DMF applications, including *Reagent*s, *Liquid*s (a volume of a reagent), *Dir*s (directions, e.g., "west" or "up"), *OnOff* states, *XYCoord*s (x-y pairs representing coordinates in a grid), etc.
- **Delayed values** (page 5), represented by *Delayed[T]* objects, are a type of *future* used to sequence operations whose point of completion is not known and which may happen in another thread. When the value has been computed, it is posted to the *Delayed* object. A thread may wait for (or join with) a future, but more often it will simply tell the future of an operation it wishes to be performed when the value is posted.
- **Change callbacks** (page 5), implemented by *ChangeCallbackList[T]* objects, are hooks on which the program can hang actions to take place when an attribute value changes. When the action is called, it will be passed both the old and new value. Examples of such hooks include liquids or wells changing reagent or volume, heaters changing current or target temperature, or pads changing electrode state associated drop. Mostly these are used to update the display, but they are also used, e.g., to monitor heater state changes to determine when a heater has reached its target temperature.
- ***Triggers*** (page 5) are an event mechanism implemented in terms of delayed values. The trigger holds a list of value/future pairs and posts the values to their respective futures when fired. *Barrier[T]*s are a type of *Trigger* that automatically fire when a fixed number of *T*s assert that they have reached the barrier. They are typically used to determine when all of a given set of drops

[1] Or as close to regular as I could get in Python/

have passed by a given point (and are so out of the way for another to move) or have finished the current phase of processing.

- ***Blobs*** (page 5) represent a *Liquid* (a *Volume* of a *Reagent*) above one or more contiguous pads (or, somewhat confusingly, a contiguous set of "on" pads that have no liquid over them). *Drops* represent the part of a blob over a given pad. The movement (including stretching, splitting, and merging) of blobs is inferred from the state of pads' electrodes and the actions of pipettors, but most of the activity in the system is a consequence of requesting that drops (especially drops in single-pad blobs) move from one pad to another.
- **Schedulable operations** (page 5) are, well, operations that can be scheduled. They include things like turning a pad on or off, dispensing a drop from a well, walking a drop from its current pad to another, and delivering a drop via an extraction point. Schedulable operations come in two flavors: *StaticOperation[V]* and *Operation[T,V]*. The difference is that the latter operations (e.g., moving a drop) require are scheduled *for* an object (or a delayed value). Scheduling an operation returns a *Delayed[V]*, and this scheduling can be chained. Additionally, when scheduling an operation, the program can request that it be delayed by a given number of ticks or amount of time. Note that schedulable operations may take time (e.g., multiple clock ticks) to perform. The futures they return are not posted until the operation is complete. Primitive schedulable operations (e.g., turning a pad's electrode on or off) are typically implemented by requesting that the *System* (or *Board*) communicate with the hardware while higher-level operations (e.g., moving a drop) are typically implemented in terms of more primitive operations.
- ***Path***s (page 5) are objects that represent a sequence of actions that a drop can take (e.g., being dispensed from a well, moving around the board, participating in mixes, being removed via an extraction point, reaching a barrier, being passed to a function). When a path is scheduled, each step is scheduled when the prior one completes. Multiple paths can be scheduled at the same time.
- **Traffic control** (page 5) is the mechanism by which drops that are moving around the board yield to one another in order to not get close enough to accidentally merge. This allows drops to, in many cases, not have to worry about other drops. The current mechanism is rather rudimentary and consists of drops *reserving* pads they wish to move to, where a drop can only be reserved if neither it nor any of its neighboring pads are reserved or contain liquid. (This reservation mechanism is also used to prevent drops from walking onto pads to which drops are being dispensed or delivered.) This mechanism is not yet sufficient to prevent deadlock in all cases, and when it is seen, other mechanisms (e.g., barriers) must be used.
- **Pipetting** (page 5) is the process of moving liquid to or from the board. It can be used to add or remove drops via extraction points as well as to add or remove liquid to and from wells. Pipetting is performed by *Pipettor* objects, which support *Supply* and *Extract* operations. Since pipetting is presumed to be asynchronous with respect to activity on the board (and since a pipettor may be

able to optimize requested transfers by reordering them), the *Pipettor* and the *PipettingTarget*s (wells and extraction points) go through a protocol in which the pipettor tells the target that it is in position and waits to be told that it is safe to perform the action, then tells the target that the action is complete (and whether the action fulfilled the request).

- **Multi-drop processes** (page 5), represented by *MultiDropProcess* objects are activities that involve multiple drops that need to be synchronized. The primary such operations in the system at present are *mixing* and *diluting* processes (processes of type *DropCombinationProcessType*) and *thermocycling* processes (processes of type *ThermocycleProcessType*). Multi-drop processes are *started* by a *lead drop*, which provides process parameters which include sufficient information to tell where all needed drops will be at the start. The other drops then *join* the process. (Note that non-lead drops may join a process that has not yet been started.) The process is responsible for moving all of the drops and signaling (via *Delayed[Drop]* objects) when the process is complete so that each drop's activity can proceed.
- ***Exerciser***s (page 5) are a framework for creating standalone programs that can be used to run *Tasks*. The *Exerciser* specifies command-line parameters that are needed to create the *System*, and each registered *Task* specifies command-line parameters needed to parameterize it, if it is the one selected to be run.
- The **monitor** (page 5), represented by a *BoardMonitor* object is a graphical user interface that serves two purposes. First, it displays a representation of the board, including the state of pads, heaters, and magnets and the location and composition of liquid, which it keeps up to date by means of change callbacks. Second it allows user interaction to (1) start, pause, and step the clock, as well as to adjust its rate, (2) turn on and off electrodes, (3) advise the system about liquid added or removed by hand, and (4) instruct the system to run statements in the macro language.
- The **macro language** (page 5) is a domain-specific programming language for DMF applications. As such, it includes data types for things like pads, wells, drops, reagents, volumes, and states, and it supports parallel operation. It is still rather primitive, though and doesn't (at the moment) support composite data (e.g., maps or lists) or looping control structures.

The system

Description of the system

The clock

Description of the clock

The engine

Description of the engine

Quantities

Description of quantities

Support types

Description of support types

Delayed values	Description of delayed values
Change callbacks	Description of change callbacks
Barriers and triggers	Description of barriers
Drops and blobs	Description of drops and blobs
Schedulable operations	Description of schedulable operations
Paths	Description of paths
Traffic control	Description of traffic control
Pipetting	Description of pipetting
Multi-drop processes	Description of multi-drop processes
Exercisers	Description of exercisers
The monitor	Description of the monitor
The macro language	Description of the macro language