Sneak Peek of Next Generation Open-RMF

Interoperability Special Interest Group
2024-05-02

Refresher on Open-RMF

- Open-RMF provides an API (C++ or Python) to hook into any mobile robot platform API
- Three flavors of API are available out of the box
  - Read Only: The mobile robot will only tell us where it is and where it intends to go
  - Traffic Light: The mobile robot tells us where it is, where it intends to go, and can be asked to pause and resume
  - Full Control: The mobile robot will tell us where it is and can obey arbitrary navigation commands
- By hooking your mobile robot into the Open-RMF API you get:
  - Traffic conflict prevention
  - Infrastructure management (e.g. automatic doors and elevators)
  - Task management (only for Full Control robots)
    - ➢ Assigning incoming task requests to robots
    - ➢ Optimal task planning
    - ➢ Automated execution of tasks

Design Motives

- Simplicity
  - The API should pass the **least** amount of information possible to and from integrators while still benefiting from all of Open-RMF's capabilities
  - It should be very clear to integrators what API to use and when
- Platform-agnostic
  - No assumptions about what protocols or commands will be used to communicate with the robot platform
  - No assumptions about how the robot performs navigation
- Stability
  - Integration work done 5 years ago should still work indefinitely into the future, even as algorithms improve and capabilities are added
  - New capabilities that require new API elements should not disrupt the behavior of earlier integrations

Early Design Choices

Implement core algorithms and API in C++

Advantages

- High performance - good for heavy computation needed by multi-agent path planning and task planning
- Popular language in robotics - supported by ROS, familiar to downstream users

Disadvantages

- Significant risk of memory errors - leads to undefined behavior and segfaults which hurt uptime and negatively impact system behaviors
- Race conditions and data races are prevalent for multi-threaded systems - we need multi-threading in order to manage multiple independent agents simultaneously

Early Design Choices

C++ PImpl (Pointer to Implementation) to receive updates from robot

```
class RobotUpdateHandle
{
public:
  void update_position(std::size_t waypoint, double orientation);
  void update_battery_soc(const double battery_soc);
  void replan();

  /* ... more methods ... */

  class Implementation;
private:
  rmf_utils::unique_impl_ptr<Implementation> _pimpl;
};
```

Advantages

- Implementation details are blocked off from the user so they can’t depend on anything that might be replaced in the future
- The API that’s meant for the user is unambiguous

Early Design Choices

C++ PImpl (Pointer to Implementation) to receive updates from robot

```
class RobotUpdateHandle
{
public:
  void update_position(std::size_t waypoint, double orientation);
  void update_position(const Eigen::Vector3d& position, const std::vector<std::size_t>& lanes);
  void update_position(const Eigen::Vector3d& position, std::size_t target_waypoint);
  void update_position(
    const std::string& map_name,
    const Eigen::Vector3d& position,
    const double max_merge_waypoint_distance = 0.1,
    const double max_merge_lane_distance = 1.0,
    const double min_lane_length = 1e-8);
  void update_position(rmf_traffic::agv::Plan::StartSet position);
};
```

Disadvantages

- Handling a variety of situations may require adding more and more functions to the public API (... e.g. 5 different ways to update the robot's position…)
- Difficult to customize behaviors since users can only touch the outer surface of the API

Early Design Choices

C++ Pure Abstract Class to push commands to robot

```
class RobotCommandHandle
{
public:
  virtual void follow_new_path(
    const std::vector<rmf_traffic::agv::Plan::Waypoint>& waypoints,
    ArrivalEstimator next_arrival_estimator,
    RequestCompleted path_finished_callback) = 0;

  virtual void stop() = 0;

  virtual void dock(
    const std::string& dock_name,
    RequestCompleted docking_finished_callback) = 0;
};
```

Advantages

- Clear indication of what the user is responsible for implementing
- Compiler guarantees that the user has implemented everything they need to

Disadvantages

- **Nothing in the interface class can be modified or added without breaking backwards compatibility**

Recent Design Choices

Callbacks to push commands to robot

```
class EasyFullControl
{
public:
  using NavigationRequest = std::function<void(Destination destination, CommandExecution execution)>;
  using StopRequest = std::function<void(ConstActivityIdentifierPtr)>;
};
```

Advantages

- Some kinds of callbacks can be made optional
- More kinds of callbacks can be added in the future without breaking prior integrations

Disadvantages

- Callbacks aren’t unified by a class which might make it harder for some users to manage the data that needs to be shared between them

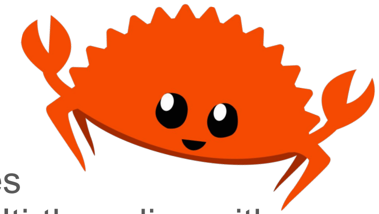
Complaints we’ve received and issues we’ve observed

- Despite being contained in just a few headers, the public API is **difficult for users to find**. And if they do find it, they might not know **how to begin** with it.
- Difficult to contribute / improve / modify the core because of the complexity and the risk of data races, undefined behavior, etc.
- Difficult to handle special situations as a user
- Path planning is not customizable enough
- Traffic negotiation is not customizable enough
- Task behaviors are not customizable enough
- Open-RMF tries to do too much
- Open-RMF doesn’t do enough

Both of these at the same time and in the same situation

Is this even solvable??

New paradigms:



- High-performance memory-safe language
  - Eliminate risk of data-races, undefined behavior, and inexplicable crashes
  - Still get maximum performance, minimal overhead, and full benefit of multi-threading with none of the risks
  - Already talked about this
- Entity Component System Design
  - Extreme modularity
  - Extreme extensibility
  - Higher performance than traditional use of interfaces and smart pointers (better CPU cache optimization)
- Service-oriented Architecture
  - System behavior is defined in terms of services which can be hierarchical
  - Users can define their own services and then inject them into the pre-existing system where they are needed
  - Custom services can build on top of pre-made services provided by Open-RMF

Entity Component System

World: A database containing and managing all data relevant to the application (some applications can choose to have multiple worlds).

Entity: General-purpose “object”. An index into data that is being stored in the World.

Component: Data that is associated with an Entity. Each Entity can have any number of Components, but only one instance of each type of Component.

System: A function that queries the World for certain combinations of components and acts upon them.

We will be using the ECS implementation from Bevy, a video game engine developed entirely in Rust.

Entity Component System

Define a custom component by making a struct and tagging it with `Component`

Declare a system by creating a function and querying for the relevant components

Use `&mut` to indicate queries that need to mutate data. `&` borrows without mutating.

Bevy will automatically run systems in parallel if they don't have conflicting `&mut`

Filter a query to only include entities whose component values changed

```rust
use bevy::prelude::*;

#[derive(Component)]
struct Kinematics {
    position: Vec3,
    velocity: Vec3,
}

#[derive(Component)]
struct Acceleration(Vec3);

#[derive(Component)]
struct Mass(f64);

#[derive(Component)]
struct Force(Vec3);

fn update_kinematics(
    mut state: Query<(&mut Kinematics, &Acceleration)>,
    time: Res<Time>,
) {
    for (mut kinematics, Acceleration(a)) in &mut state {
        kinematics.velocity += *a * time.delta_seconds();
        let v = kinematics.velocity;
        kinematics.position += v * time.delta_seconds();
    }
}

fn update_acceleration(
    mut dynamics: Query<
        (&mut Acceleration, &Mass, &Force),
        Or<(Changed<Force>, Changed<Mass>)>,
    >,
) {
    for (mut acceleration, Mass(m), Force(f)) in &mut dynamics {
        acceleration.0 = *f / *m;
    }
}
```

Entity Component System

Define an application by making a new **App** and adding systems to it.

If it is important for one system to always run after another in each update cycle, you can specify that with `.after(_)`.

To make it easy to reuse or distribute application logic, encapsulate it in a plugin.

Then users can add it to their own application using a single `.add_plugins(_)`.

```
use bevy::prelude::*;

/* ... Component and system definitions here ... */

fn main() {
    let mut app = App::new();
    app.add_systems(Update, (
        update_acceleration,
        update_kinematics.after(update_acceleration),
    ));
    app.run();
}
```

```
use bevy::prelude::*;

/* ... Component and system definitions here ... */


pub struct PhysicsPlugin;
impl Plugin for PhysicsPlugin {
    fn build(&self, app: &mut App) {
        app.add_systems(Update, (
            update_acceleration,
            update_kinematics.after(update_acceleration)
        ));
    }
}

fn main() {
    let mut app = App::new();
    app.add_plugins(PhysicsPlugin);
    app.run();
}
```

Service-oriented Architecture

Application behaviors are broken down into discrete “services”, each having a well-defined input, output, and effect on the world.

Bevy supports Systems that can take an input argument and produce an output value.

We can use this to develop **Systems as Services**.

These Services can then be chained together to define larger Services.

Path Planner
Input: Robot, Destination
Output: Path

Navigation
Input: Robot, Path
Stream: Progress
Output: Arrival / Failure

Elevator
Input: Name, Arrival Floor
Stream: Arrival Progress
Output: Session Manager

Clean
Input: Robot, Zone
Stream: Progress
Output: Finished / Failure

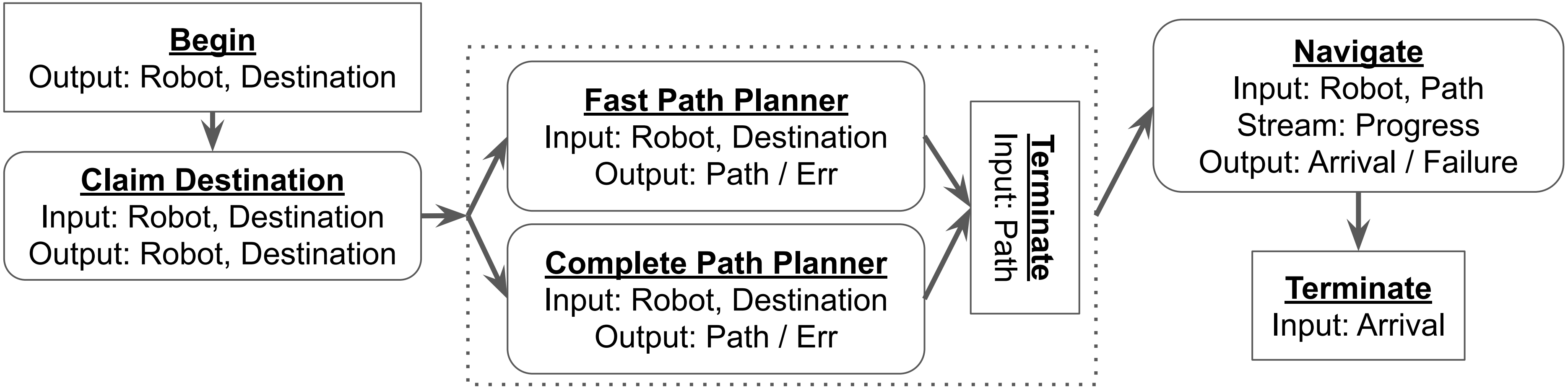
Service-oriented Architecture

Services can be run in parallel and synced at relevant points.

Fast Path Planner finds simple paths quickly
Complete Path Planner finds difficult paths eventually

Race them against each other and send the winner to the motion planner.

```
fn go_to_place(scope, builder) {
    scope.input.chain(builder)
    .then(claim_destination)
    .then_scope(|scope, builder| {
        scope.input.chain(builder)
        .fork_clone((
            |branch|
                branch.then(fast_planner).terminate(),
            |branch|
                branch.then(complete_planner).terminate(),
        ))
    })
    .then(navigate)
    .terminate();
}
```



Service-oriented Architecture

Users can define their own services by implementing Bevy systems with inputs and outputs

- ➢ Besides the inputs and outputs of the service definition, these system-services can view and modify anything in the World while running

Services can be defined by chaining together other services

Hierarchical services can support dependency injection, so users can insert custom services into premade service hierarchies

Open-RMF will define out-of-the-box services and service hierarchies which users can opt-in / opt-out of and customize as they see fit

How we will achieve our design goals

Simplicity / Ease-of-use	We will provide high-level Bevy Plugins that work out of the box with minimal configuration for common use cases.
Covering more use cases	We will provide more granular plugins that can be selected by users as needed, tailored to more specific use cases.
Stability	Released plugins will remain API-compatible indefinitely into the future. New capabilities that need new APIs will be introduced through new plugins.
Easier to contribute	The modularity of Bevy's ECS and our service architecture will help isolate units of capability to make it easier for contributors to introduce new capabilities or improve existing ones.
Customizability	Users can define and inject their own services into premade plugins and systems.
Accessibility	We will continue to provide C++ and Python APIs that wrap the Rust implementation. Users will be able to inject services that are implemented in C++ and Python.

Long term plans

In the long-term we intend to support the ability for users to sketch workflow graphs in a GUI to define how their robots should behave, e.g. how a task should be executed. Each node in the graph refer to a service, using bevy_impulse to tie the services together and execute the workflow.